# Theory of 3-4 Heap

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in the
University of Canterbury
by
Tobias Bethlehem

---

**Examining Committee**

| | |
|---|---|
| Prof. Tadao Takaoka | Supervisor |
| Prof. Tomio Hirata | Examiner |

University of Canterbury
2008

To my lovely wife

# Acknowledgments

First and foremost I would like to thank my wife for her understanding and patience, without her support this thesis would not be a reality.

I also want to thank my supervisor, Prof. Tadao Takaoka, for his guidance, knowledge and humour. Without his support and original research of the 2-3 heap, this thesis would never have been possible.

# Abstract

As an alternative to the Fibonacci heap, and a variation of the 2-3 heap data structure by Tadao Takaoka, this research presents the 3-4 heap data structure. The aim is to prove that the 3-4 heap, like its counter-part 2-3 heap, also supports $n$ insert, $n$ delete-min, and $m$ decrease-key operations, in $O(m + n \log n)$ time. Many performance tests were carried out during this research comparing the 3-4 heap against the 2-3 heap and for a narrow set of circumstances the 3-4 heap outperformed the 2-3 heap.

The 2-3 heap has got a structure based on dimensions which are rigid using ternary linking and this path is made up of three nodes linked together to form a trunk, and the trunk is permitted to shrink by one. If further shrinkage is required then an adjustment is made by moving a few nodes from nearby positions to ensure the heaps rigid dimensions are retained. Should this no longer be the case, then the adjustment will trigger a make-up event, which propagates to higher dimensions, and requires amortised analysis. To aid amortised analysis, the trunk is given a measurement value called potential and this is the number of comparisons required to place each node into its correct position in ascending order using linear search.

The divergence of the 3-4 heap from the 2-3 heap is that the trunk maximum is increased by one to four and is still permitted to shrink by one. This modified data structure will have a wide range of applications as the data storage mechanism used by graph algorithms such as Dijkstra's 'Single Source Shortest Path'.

Table of Contents

# List of Figures

# List of Tables

# Chapter I

# Introduction

## *1.1 Topic*

Based upon Takaoka's 2-3 heap [1] research as an alternative to the Fibonacci heap [4], this research document will describe and formalise the 3-4 heap as an alternative to the 2-3 heap. The objective of this research is to prove that the 3-4 heap also supports $n$ insert, $n$ delete-min, and $m$ decrease-key operations, in $O(m + n \log n)$ time and to discover under which operating conditions the 3-4 heap is better or worse than the 2-3 heap.

A variety of experiments will be performed using the data structures in stand-alone form through a test harness and as the data storage mechanism used by established graph algorithms such as Dijkstra's [2] 'Single Source Shortest Path'. The data structure can also be used for Prim's [3] 'Minimum Cost Spanning Tree'.

The performance of the heaps will be measured using two key indicators; the number of key comparisons performed during each experiment and time complexity by measuring how much time in milliseconds has elapsed for each experiment.

To ensure the results of each experiment were comparable between the 2-3 heap and 3-4 heap, both heaps were required to be under identical operating and data conditions. This has been achieved by ensuring both heaps share the same Application Programming Interface (API) and therefore can be interchanged dynamically without exiting the application.

One practical real world application for the 3-4 heap is to be used as the data storage mechanism for Dijkstra's 'Single Source Shortest Path' used by hand held Global Position Systems (GPS) devices.

### *1.2 Background*

The Fibonacci heap is a data structure that is more complicated than an ordinary heap and is asymptotically faster for some operations. The Fibonacci heap is a collection of heap-ordered trees. The key value of a child node is always greater than or equal to the key value of its parent node and the minimum key valued node will always be at the root of one of the trees [10]. The potential of a Fibonacci heap is determined by the number of trees.

The Fibonacci heap shares three operations with the 2-3 heap, these are insert, delete-min, and decrease-key. The insert operation works by creating a new Fibonacci heap containing one tree with one node and then performs a merge operation between two Fibonacci heaps. Merging is performed by binary linking two trees of the same degree together repeatedly, any carries can cause additional binary linking. The delete-min operation is performed in four steps. Step one removes the node with the minimum key value from the top level position. Step two inserts into the top level position all the children of the removed node. Step three merges together all trees with the same degree and step four updates the pointer to the minimum key valued node. The decrease-key operation is performed in three steps. Step one reduces the key value of a node and makes sure its new value is still greater than or equal to its parent node. Otherwise in step two the node is removed and inserted into the top level position and followed by step three which will restructure the tree the node was removed from.

The 2-3 heap has got a structure based on dimensions which are rigid using ternary linking, but not as rigid as the binomial queue invented by Vuillemin [5]. This path is made up of three nodes linked together to form a trunk, and the trunk is permitted to shrink by one, hence, 2-3 heap. If there is a requirement to shrink further, an adjustment is made by moving a few nodes from nearby positions to ensure the heaps rigid dimensions are retained. Should this no longer be the case, then the adjustment will trigger a make-up event, which propagates to higher dimensions, and will also require amortised analysis.

To aid amortised analysis, the trunk is given a measurement value called 'potential', used in reverse meaning as used in [4]. This potential is used to

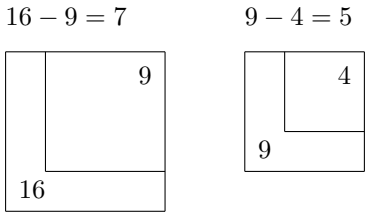


Figure 1.1: 3-4 heap and 2-3 heap maximum and minimum workspace size

describe the real potential of a tree by counting the number of comparisons required to place each node into its correct position in ascending order, this position is found by using linear search. We define the potential of a trunk with one node to be zero, two nodes to be one and three nodes to be three. The potential of the 2-3 heap is the sum of those potentials.

Each node is valued by a label. The smallest valued node in a heap is called the minimum node. A 2-3 heap can be constructed by ternary linking the root node of each tree of the same degree, where the degree of a tree is the number of children nodes beneath the root. The 2-3 heap supports $n$ insert, $n$ delete-min, and $m$ decrease-key operations, in $O(m + n \log n)$ time. These three operations allow the following operations to be performed on a heap: the insertion of node into heap, the reduction of a node's key value and the deletion of the smallest key valued node.

The divergence of the 3-4 heap from the 2-3 heap is that the trunk maximum is increased by one to four yet still remains permitted to shrink only by one, thus this also increased the minimum size of the trunk by one to three. The potential of a trunk with four nodes is defined to be six. Figure 1.1 presents the differences in workspace size of these two heaps in graphical form. A workspace can contain up to sixteen (16) nodes on the same layer, see Section 2.1 for a precise definition of workspace. With all trunks at their minimum size, a single 3-4 heap workspace contains nine (9) nodes and at its maximum it has sixteen (16) nodes. The workspace buffer is therefore seven (7) nodes, two (2) more than the 2-3 heap.

Using the 2-3 heap as the measurement base line, the motivation of this research is to measure the change in performance by having a larger workspace buffer and increasing the length of the trunk by one node and their impact

on the operations of insert, delete-min and decrease-key.

# Chapter II

# Definitions and Theory

## *2.1 Definitions of Terminology*

This section defines the terminology used to identify and reference a 3-4 heap instance. Table 2.1 contains important heap terminology and these are highlighted in Figure 2.1 and Figure 2.6.

In Figure 2.2, the 3-4 heap trees are both of degree two (Section 2.2) and display the minimum and maximum rigid trunk dimensions. The left-most is a complete sixteen node tree with potential (Section 2.5) of thirty, and the right-most is a complete nine node tree with a potential of twelve, these form the upper and lower bounds of standard arrangement and permit the degree two tree to grow and shrink by seven (7) nodes.

The trunk sloping left-down (towards 7 o'clock) is in the 1st dimension and the trunk sloping right-down (towards 5 o'clock) is in the 2nd dimension. If this structure is part of a larger 3-4 heap tree where each node itself represents a sub-tree, those trunks can be said to be of $i$-th dimension and $(i+1)$-th dimension. This can be viewed in the heap labelled $D$ in Figure 2.6.

A workspace is defined as being all nodes located within the $i$-th dimension where a tree operation, such as insert or decrease-key, is in progress and

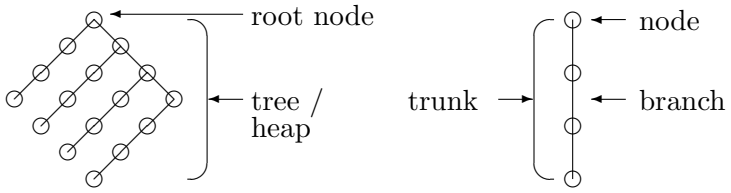


Figure 2.1: Heap terminology

| Term | Description |
|---|---|
| node | This is the circle and is used to represent a key value with a label attached. |
| branch | This is the line connecting two nodes and a branch of trunk length of one. |
| root node | This is the node with no parent node. |
| trunk | A trunk is a straight line connection several nodes. |
| main trunk | Like a trunk, but this connects together up to three trees/heaps at root level. See Figure 2.6. |
| tree | This is a group of trunks connected together without any labels. |
| labelled tree | This is a group of trunks connected together with labels attached. |
| heap | This is a group of labelled trees with numerical values in correct order. |

Table 2.1: Important heap terminology

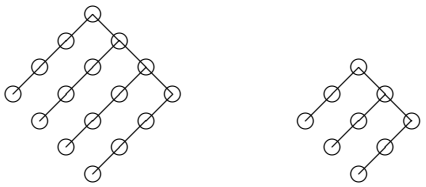

Figure 2.2: A 3-4 heap with maximum and minimum number of nodes

a single trunk in the $(i+1)$-th dimension which the $i$-th dimension trunks are part of. The workspace does not extend to other sub-trees which also reside in the $i$-th dimension.

Each trunk on the $i$-th dimension can have at minimum three nodes and a maximum of four nodes, hence the term 3-4 heap. During each operation on the tree workspace, these operations can result in a breach of the minimum/maximum number of nodes per trunk. When this occurs the workspace will require to be reshuffled back into standard arrangement before the heap operation can be completed.

Each node in the 3-4 heap will be assigned a numerical value, the node with the smallest label is at the root position and each subsequent node on the trunk traversed in ascending order allowing for equal keys.

### *2.2 Degree of a Tree*

The degree of a tree relates to how many branches connect off the root node. By looking at Figure 2.6, we can see that each tree labelled $A$, $B$, $C$ and $D$ has got a different degree. Label $A$ is of degree zero because there are no branches connected to its root node. Label $B$ is of degree one because there is one branch connected to its root node. Label $C$ is of degree two because there are two branches connected to its root node. Label $D$ is of degree three because there are three branches connected to its root node. It can also be observed that trees of higher degrees contain those of lower degrees in their structure, for example, label $C$ is a degree two tree and is comprised of three trees of degree one.

### *2.3 Definition of a Tree*

This section will define a tree in mathematical terms by using terms defined for the 2-3 heap by Tadao Takaoka [1]. A polynomial of complete r-ary trees is defined [1] as:

$$P = \mathbf{a}_{k-1}\mathbf{r}^{k-1} + \cdots + \mathbf{a}_1\mathbf{r} + \mathbf{a}_0 \tag{2.1}$$

Here $\mathbf{r}$ is a linear tree of size r. The product of linear trees $\mathbf{a}$ and $\mathbf{b}$ is the

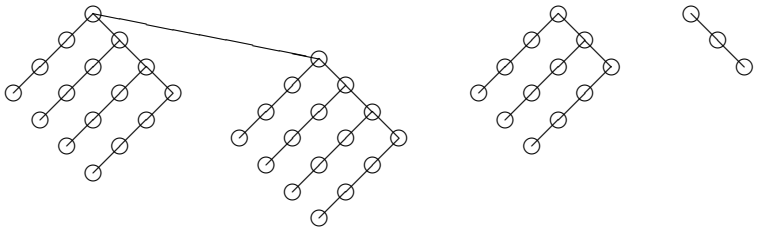

Figure 2.3: An r-ary tree

direct product of **a** and **b** where corresponding roots are connected.

In (2.1), $P$ is the polynomial, $\mathbf{a}_i (0 \leq i \leq k-1)$ is a linear tree co-efficient and the size of $\mathbf{a}_i$ is between zero (0) and $r-1$. The "+" means the addition in a collection of trees. The term $\mathbf{r}^i$ represents a complete r-ary tree of degree up to $k-1$ where each trunk always has the maximum number of nodes. For the tree to stay in its current position, the length of $\mathbf{a}_i$ must be at most $r-1$. If $\mathbf{a}_i$ is $\mathbf{r}$, then this is a carry because it makes $\mathbf{a}_i\mathbf{r}^i \Rightarrow \mathbf{r}^{i+1}$.

There are two parts to $\mathbf{a}_i\mathbf{r}^i$ and these are $\mathbf{a}_i$ and $\mathbf{r}^i$. Part $\mathbf{r}^i$ represents a tree of size $r^i$. Text emphasis is used to distinguish tree size from number of nodes. On the 3-4 heap, $\mathbf{r}$ represents the maximum trunk length of four (4). Inspecting the left-most tree in Figure 2.3, this is a tree of degree three (3) so by definition $k-1=2$. Since $\mathbf{r}^i$ can only represent a single tree in the $1^{st}$ and $2^{nd}$ dimension, it can only represent half of the left-most tree. Putting all of this together; $\mathbf{r}^i$ represents a tree of $\mathbf{4}^2$ and its node count is $4^2 = 16$. Part $\mathbf{a}_i$ represents the number of trees of $\mathbf{r}^i$ there are on the main trunk. On the main trunk in the left-most tree of Figure 2.3, there are two trees of $\mathbf{r}^i$ making $\mathbf{a}_i = 2$ through the definition of $k-1=2$. By combining $\mathbf{a}_i$ and $\mathbf{r}^i$, this tree is represented by $\mathbf{2} \times \mathbf{4}^2$ and the node count of this tree is $2 \times 4^2 = 32$.

Using formula (2.1) above the three different sized trees in Figure 2.3 can be represented as $\mathbf{2} \times \mathbf{4}^2 + \mathbf{3} \times \mathbf{4} + \mathbf{3}$ and their total node count is $2 \times 4^2 + 3 \times 4 + 3 = 47$.

The definition for r-ary trees requires some relaxing in order to accommodate trunk shrinkage which is allowed for the 2-3 heap and 3-4 heap. So the following is defined [1]:

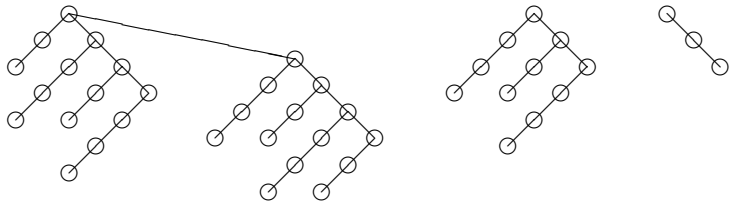

Figure 2.4: A relaxed r-ary tree

$$\begin{aligned} T(0) &= \text{a single node} \\ T(i) &= T_1(i-1) \bullet \cdots \bullet T_m(i-1) \ (m \text{ is between } l \text{ and } r) \end{aligned} \tag{2.2}$$

Where $l$ is the minimum number of nodes possible on a trunk, and $r$ is the maximum number of nodes possible on a trunk. Symbol $T$ represents a tree and the tree is permitted to be incomplete in definition by not always having the maximum number of nodes per trunk. For the 3-4 heap, $l = 3$ and $r = 4$. The coefficient of $T$ is the number of nodes making up the main trunk and $i$ is the number of branches connected to the root node except for the main trunk. Symbol "$\bullet$" represents an operation defined by $L = S \bullet T$ where $S$ and $T$ are trees and $L$ is a new tree by linking $S$ and $T$ such that the root node of tree $T$ is the child of the root node of tree $S$. The left-most tree in Figure 2.4 displays this linkage of $S$ and $T$ with tree $S$ being on the left and tree $T$ on the right. Expression (2.2) is always evaluated right to left.

The definition for the formula used to represented a relaxed r-ary tree is defined in (2.3) by combining (2.1) and (2.2) together as follows [1]:

$$P = \mathbf{a}_{k-1}T(k-1) + \cdots + \mathbf{a}_1 T(1) + \mathbf{a}_0 \tag{2.3}$$

The trees in Figure 2.4 display a relaxed r-ary tree of $\mathbf{2}T(2) + \mathbf{3}T(1) + \mathbf{3}T(0)$. Note that $T(i)$ is a type of tree whose degree is $i$.

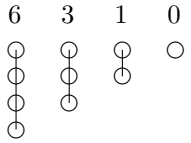


Figure 2.5: Measuring trunk potential

### *2.4 Top Level Workspace*

The top level workspace is best described as being constructed from a row of pigeon holes. Each pigeon hole is uniquely designated to hold trees of one particular degree connected by a main trunk, starting with the smallest pigeon hole holding a tree connecting a few of trees of degree zero, that is, a few nodes, and incrementing consecutively higher by one degree. Each pigeon hole therefore will hold trees of type $T(i)$ connected by a main trunk where $i \geq 0$.

When the number of nodes in a tree grows or shrinks, the tree may no longer remain within standard arrangement for its respective tree degree. The tree must now be carried into another top level position. This means that the tree which is currently occupying $T(i)$ is removed from its position and merged into any existing tree already of type $T(i+1)$ or $T(i-1)$, as determined by the tree's new degree level. The algorithm used during process is detailed in Section 2.8.2. Dijkstra's algorithm will always carry to top level position $T(i+1)$.

### *2.5 Tree Potential*

The potential of a tree is based upon summing the potential measured on each trunk. The formula used to determine the potential of a trunk is based upon the number of linear comparisons (worst case) that are required to have each node positioned in ascending order allowing for equal keys. See Figure 2.5 for a graph detailing the potential for varying trunk lengths. It should be noted that whilst each node making up this workspace could be the root node of another workspace at another depth, these additional workspaces are not taken into consideration. Only the nodes within the current workspace are

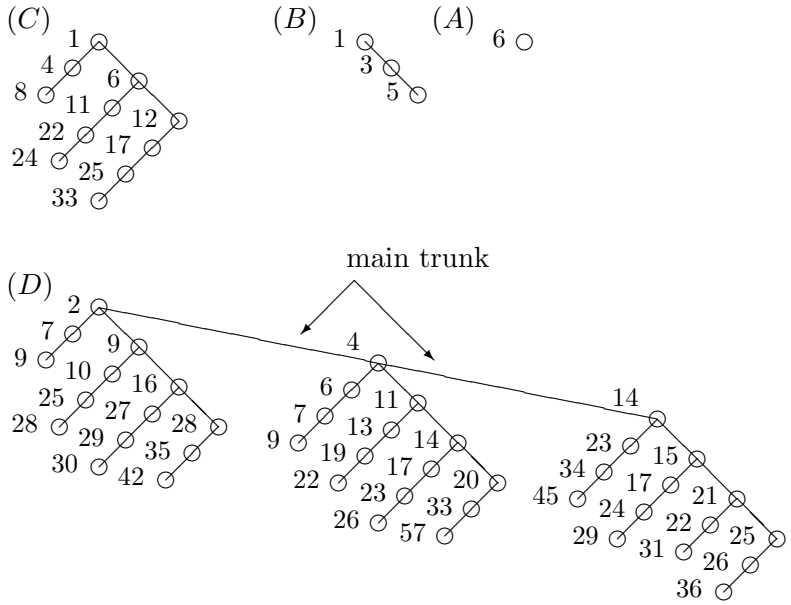


Figure 2.6: Four sample 3-4 heaps

considered because the potential of other parts does not change before and after an operation.

If a more efficient algorithm such as comparison algorithm $B$ in Section 2.7 was used, the potential of the four node trunk would have reduced by one (1) to five (5). Unfortunately having the four node trunk defined with a potential of five (5) would have resulted in breaking one of the most important requirements of the make-up process, that is for its amortised cost to equal zero. For precise definition of make-up process see Section 2.8.1 and see next section for a precise definition of amortised cost. By having the four node trunk potential defined as six (6), the make-up process amortised cost does equal zero.

## *2.6 Amortised Cost*

Amortised cost is a method which gives a quantifiable and standardised measure to all tree operations. It uses two core features of the tree; first is the potential of the tree before and after a tree operation, and secondly, the number of node to node key comparisons used during that operation. We

measure the cost of computation by the number of comparisons.

The formula used to calculate the amortised cost figure is to take the tree potential before an operation commences and add the number of key comparisons used. Next subtract from this figure the tree potential after the operation has completed. The amortised cost of the operation is calculated. There are three possible outcomes for the resultant figure:

1. If positive, then a cost was incurred during that tree operation

2. If zero, then no cost was incurred during that tree operation and it was essentially free

3. If negative, then a profit was incurred during that tree operation

Of the three possible outcomes, having a negative amortised cost is of most interest. There is only one scenario where the amortised cost of an operation can result in a negative cost and this is when the number of node-to-node comparisons used is less than the growth in tree potential after the completion of an operation. An example of this occurrence is with the insertion of a single node into a three node trunk, it will take up to three linear search comparisons to insert this node into its correct trunk position. The post-operation potential of the trunk has increased by three from three to six and the amortised cost is zero. But by using an optimised insertion algorithm (Section 2.7 algorithm $B$) this operation can be performed using only two comparisons. The final trunk potential still remains six because it is calculated using linear search (worst case), but the actual number of comparisons used is one less than if linear search was used, and this makes the amortised cost of this insert operation negative one, a profit.

Amortised cost analysis during the $i$-th operation is defined as follows:

$$\begin{aligned} a_1 &= t_1 + (\Phi_0 - \Phi_1) \\ a_2 &= t_2 + (\Phi_1 - \Phi_2) \\ &\vdots \\ a_i &= t_i + (\Phi_{i-1} - \Phi_i) \end{aligned}$$

Where $a_i$ is the amortised cost of an operation, $t_i$ is the actual number of comparisons, $\Phi_{i-1}$ is the current potential, $\Phi_i$ is the potential after the operation. At the conclusion of all operations the above reduces to:

$$A = T + (\Phi_0 - \Phi_N)$$

Where $A$ is the total amortised cost, $T$ is the total cost and $N$ is the final step. Since $\Phi_0$ is the starting potential of zero and if $\Phi_N$ is zero then the finishing potential will be zero at end of the $N$-th operation, this reduces to:

$$A = T$$

If the end potential is zero, then $n$ insert, $n$ delete-min and $m$ decrease-key operations can be done in $O(m + n \log n)$ time.

### *2.7 Performing Node Comparisons*

The following algorithm makes up the foundation of nearly all comparisons used within all workspace and top-level insertions. Standard linear ascending comparisons are achieved by starting with the smallest node on a trunk and comparing it with the node which requires insertion. If the insertion position is not found, then the next node on the trunk is compared against the node which requires insertion, and so forth.

Looking at Figure 2.7, it should be noted it contains sub-diagrams $A$, $B$ and $C$. Each of these sub-diagrams respectively covers a special comparison algorithm where the number of comparisons required were minimised while potential was maximised, thus minimising amortised cost or ensuring the amortised cost equalled zero. The comparison algorithms used are as follows:
Comparison algorithm for $A$:

Starting with two two-node trunks, three comparisons would be required in the worst case situation to merge these two together. The black nodes found at each step are the two nodes getting compared. In the following step, either a full four node trunk has been created, or, the two black nodes which were just compared together have got a joining line and another two nodes have been marked with black as being the next pair to compare.
Comparison algorithm for $B$:

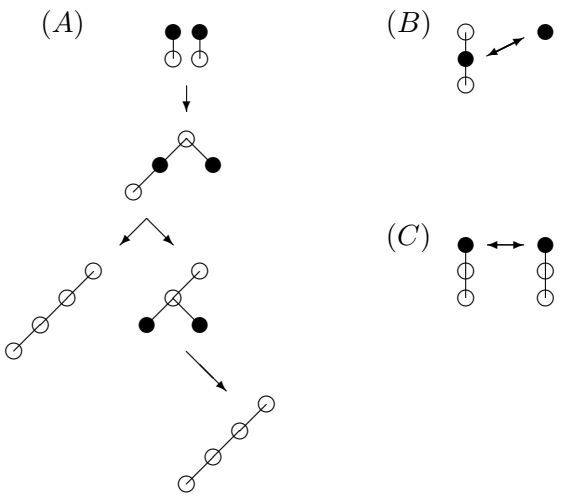


Figure 2.7: Special node comparison algorithms

With one three-node trunk and a single node, up to three comparisons would be required using linear comparisons before the correct insertion point for the single node in the three node trunk was found. Using a more optimised approach, this can be reduced to two comparisons, saving one comparison, but this approach requires a constant two comparisons while linear comparisons may require one, two or sometimes three comparisons. However, it is the maximum number of comparisons required, not the average, which is used when determining the number of comparisons required.

The optimal algorithm approach is executed by comparing the single node with the middle node on the three node trunk. This will determine if the second node to be compared will be the top-most or bottom-most node. After comparison with the second node, its final insertion position has been determined.

Comparison algorithm for $C$:

With two three-node trunks, the most optimal algorithm to create a four node trunk is to compare their top-most nodes using one comparison. Having now established which of these two nodes is the smallest, remove it from its trunk and insert into the top-most position of the other trunk. This process results in a four node and a two node trunk.

## *2.8 Detailed Description of Operations*

### *2.8.1 Decrease-key*

The following series of figures display what happens when we decrease the number of nodes in the 3-4 heap workspace by selecting any of the black nodes. In the following figures, only the current workspace of $i$-th and $(i+1)$-th dimension is shown because each node can be regarded as a tree of the same degree. When the value of a node is decreased it is removed from its workspace. It should be noted that whilst each node making up this workspace could be the root node for another workspace in another depth, these additional workspaces are by association also removed from the tree. The left-hand side and right-hand side tree structures in each figure represent the before and after tree manipulations. The workspace potential is measured only before and after the change, and the workspace potential which resides in other dimensions remains unchanged and is excluded from these calculations. Before the completion of tree manipulations, it must be ensured that the new shape of the tree remains within standard arrangement. Each figure is also accompanied with a description of what happened and has an accompanying table of figures to portray important statistics. To aid in describing what happens in each figure, labels $a$, $b$, $c$, ..., are attached to the trees to identify trunks or nodes.

Following is the legend description for the series of tables used in this section detailing important figures about each scenario case:

| **Legend** | **Description** |
|---|---|
| Count of nodes | Count of nodes present in left tree |
| Start potential | Measured potential of left tree |
| End potential | Measured potential of right tree |
| Comparisons | Number of comparisons used to transform the left tree into its new shape on the right |
| Amortised | Sum value made up from difference between start and end potential plus number of comparisons spent |

Case 1: Refer to Figure 2.8.

Removing any one of the twelve black nodes will bring the tree new shape

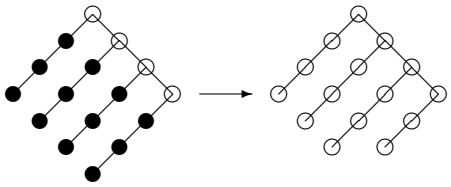

Figure 2.8: Decrease-key case 1

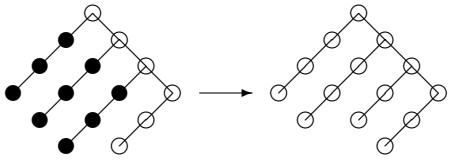

Figure 2.9: Decrease-key case 2

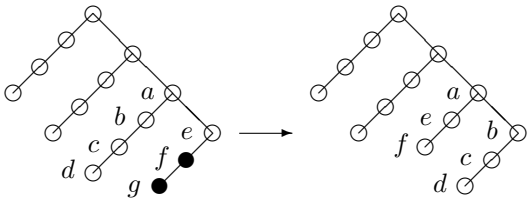


Figure 2.10: Decrease-key case 3

within standard arrangement on the right. The potential is decreased by three with no comparisons spent.

| | | | |
|---|---|---|---|
| Count of nodes | 16 | | |
| Start potential | 30 | End potential | 27 |
| Comparisons | 0 | Amortised | 3 |

Case 2: Refer to Figure 2.9.

Removing any one of the nine black nodes will bring the tree new shape within standard arrangement on the right. The potential is decreased by three with no comparisons spent.

| | | | |
|---|---|---|---|
| Count of nodes | 15 | | |
| Start potential | 27 | End potential | 24 |
| Comparisons | 0 | Amortised | 3 |

Case 3: Refer to Figure 2.10.

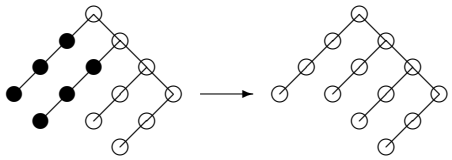

Figure 2.11: Decrease-key case 4

Removing any one of the two black nodes will cause the tree new shape to violate the standard arrangement rules; the trunk has only two nodes but should contain three or four. The tree therefore requires to be restructured in the most efficient means to retain as much potential as possible and to keep comparisons to an absolute minimum. In this particular case, we can take advantage of the heap definition requiring nodes to be placed in ascending order. In this example figure, node $(g)$ has been removed and to restructure the heap the following was done: simply swap nodes $(b)$, $(c)$, and $(d)$ with nodes $(e)$ and $(f)$, no comparisons were required and the tree is no longer violating the standard arrangement rules. The new standard arrangement is on the right. The potential is decreased by three with no comparisons spent.

| | | | |
|---|---|---|---|
| Count of nodes | 15 | | |
| Start potential | 27 | End potential | 24 |
| Comparisons | 0 | Amortised | 3 |

Case 4: Refer to Figure 2.11.

Removing any one of the six black nodes will bring the new shape within standard arrangement on the right. The potential is decreased by three with no comparisons spent.

| | | | |
|---|---|---|---|
| Count of nodes | 14 | | |
| Start potential | 24 | End potential | 21 |
| Comparisons | 0 | Amortised | 3 |

Case 5: Refer to Figure 2.12.

Removing any one of the four black nodes will cause the new shape to violate the standard arrangement rules. This is resolved by taking advantage of the non-increasing order rule of the nodes. The same restructuring process can be used regardless of which one of the four black nodes was removed. In this particular example, node $(m)$ has been removed and to restructure

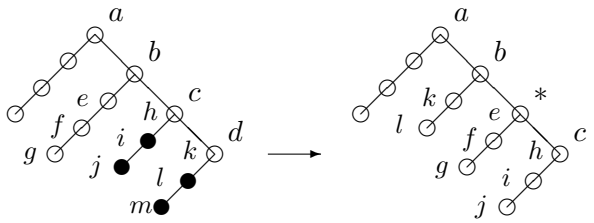


Figure 2.12: Decrease-key case 5

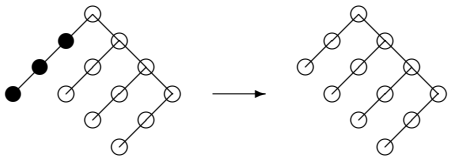

Figure 2.13: Decrease-key case 6

the following was done: remove nodes $(e)$, $(f)$ and $(g)$ from trunk $b$, delete trunk $d$ by removing nodes $(k)$ and $(l)$ and insert them beneath trunk $b$ using no comparisons. Using one comparison, compare node $(e)$ with node $(h)$ and create a new trunk in the correct $(i+1)$-th dimension position using nodes $(e)$, $(f)$, and $(g)$. In this example the value of node $(e)$ was smaller than node $(h)$ so trunk $c$ was displaced by the new trunk '$*$'. The new standard arrangement is on the right. The potential is decreased by three with one comparison spent.

*Note: The previous case started with and resulted in the same shaped tree, but the methodology used and its amortised cost are different.*

| | | | |
|---|---|---|---|
| Count of nodes | 14 | | |
| Start potential | 24 | End potential | 21 |
| Comparisons | 1 | Amortised | 4 |

Case 6: Refer to Figure 2.13.

Removing any one of the three black nodes will bring the new shape within standard arrangement on the right. The potential is decreased by three with no comparisons spent.

| | | | |
|---|---|---|---|
| Count of nodes | 13 | | |
| Start potential | 21 | End potential | 18 |
| Comparisons | 0 | Amortised | 3 |

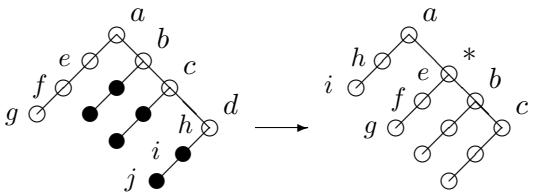


Figure 2.14: Decrease-key case 7

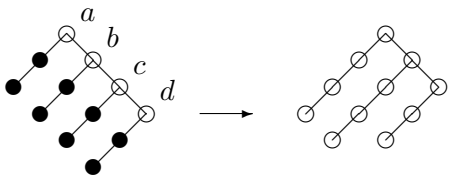


Figure 2.15: Decrease-key case 8

Case 7: Refer to Figure 2.14.

Removing any one of the six black nodes will cause the new shape to violate the standard arrangement rules. This is resolved by taking advantage of the non-increasing order rule of the nodes. The same restructuring process can be used regardless of which one of the six black nodes was removed. In this particular example, node $(j)$ was removed and to restructure the following was done: remove nodes $(e)$, $(f)$ and $(g)$ from trunk $a$, delete trunk $d$ by removing nodes $(h)$ and $(i)$ and insert them beneath trunk $a$ using no comparisons. Using two comparisons, compare node $(e)$ with the $(i+1)$-th dimension nodes of trunks $b$ and $c$, and create a new trunk in the correct $(i+1)$-th dimension position using nodes $(e)$, $(f)$ and $(g)$. In this example the value of node $(e)$ was smaller than the $(i+1)$-th dimension nodes comprising of trunks $b$ and $c$, so trunks $b$ and $c$ were displaced by the new trunk '$*$'. The new standard arrangement is on the right. The potential is decreased by three with two comparisons spent.

*Note: The previous case started with and resulted in the same shaped tree, but the methodology used was different as were their amortised cost.*

| | | | |
|---|---|---|---|
| Count of nodes | 13 | | |
| Start potential | 21 | End potential | 18 |
| Comparisons | 2 | Amortised | 5 |

Case 8: Refer to Figure 2.15.

Removing any one of the eight black nodes will cause the new shape to violate the standard arrangement rules. The process to resolve the tree back into standard arrangement depends on which trunk is violating it and these are as follows:

Trunk $a$:

Remove the two nodes from $i$-th dimension on trunk $b$ and create a four node trunk by inserting these into trunk $a$ using two comparisons. Next swing across the three nodes of trunk $c$ to beneath trunk $b$, creating a four node trunk using zero comparisons. A total of two comparisons were spent.

Trunks $b$ and $c$:

On the trunk which lost the node, remove the remaining black node and insert this into trunk $a$ using two comparisons with its $i$-th dimension nodes. If the node was removed from trunk $b$, then swing across all three nodes from trunk $c$, thus creating a new four-node trunk with zero comparisons spent. If the node was removed from trunk $c$ then swing across all three nodes from trunk $d$, thus creating a new four-node trunk with zero comparisons spent. A total of two comparisons were spent.

Trunk $d$:

From trunk $c$ reduce the nodes in $i$-th dimension by one and insert this node into trunk $d$ using two comparisons. With trunk $c$ now having the shortest $i$-th dimension length, restructure workspace as described above for trunk $c$. A total of four comparisons were spent. An alternative approach would be to remove both of the remaining nodes from trunk $d$ and insert each of its nodes respectively into trunks $a$ and $b$. Two comparisons are used to insert each node. A total of four comparisons were spent.

Even though trunks $a$–$c$ used only two comparisons, trunk $d$ used four comparisons and the worst case amortised cost is recorded.

| | | | |
|---|---|---|---|
| Count of nodes | 12 | | |
| Start potential | 18 | End potential | 18 |
| Comparisons | 4 | Amortised | 4 |

Case 9: Refer to Figure 2.16.

Removing any one of the nine black nodes will bring the new shape within standard arrangement on the right. The potential is decreased by three with

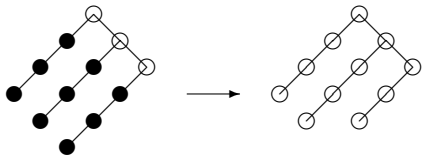

Figure 2.16: Decrease-key case 9

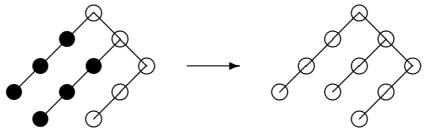

Figure 2.17: Decrease-key case 10

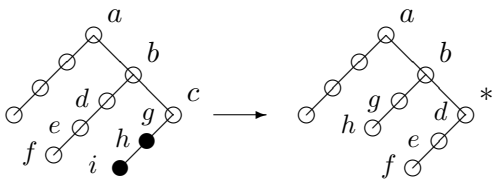


Figure 2.18: Decrease-key case 11

no comparisons spent.

*Note: The previous case resulted in the same shaped tree, but the starting shape, algorithm used, and amortised cost are different.*

| | | | |
|---|---|---|---|
| Count of nodes | 11 | | |
| Start potential | 21 | End potential | 18 |
| Comparisons | 0 | Amortised | 3 |

Case 10: Refer to Figure 2.17.

Removing any one of the six black nodes will bring the new shape within standard arrangement on the right. The potential is decreased by three with no comparisons spent.

| | | | |
|---|---|---|---|
| Count of nodes | 11 | | |
| Start potential | 18 | End potential | 15 |
| Comparisons | 0 | Amortised | 3 |

Case 11: Refer to Figure 2.18.

Removing any one of the two black nodes will cause the new shape to violate the standard arrangement rules. This is resolved by taking advantage

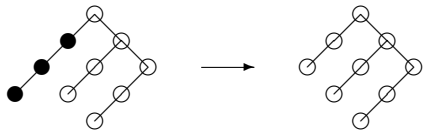

Figure 2.19: Decrease-key case 12

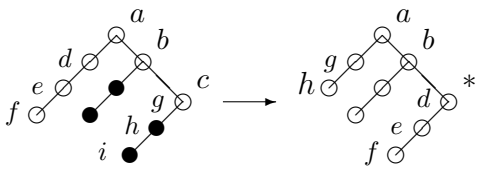


Figure 2.20: Decrease-key case 13

of the ascending order for nodes. In this particular example, node $(i)$ was removed and to restructure the following was done: remove nodes $(d)$, $(e)$ and $(f)$ from trunk $b$, delete trunk $c$ by removing nodes $(g)$ and $(h)$ and insert them beneath trunk $b$ using no comparisons. Using no comparisons, create a new trunk '$*$' in the same $(i+1)$-th dimension position where trunk $c$ used to be with nodes $(d)$, $(e)$ and $(f)$. The new standard arrangement is on the right. The potential is decreased by three with no comparisons spent.

*Note: The previous case started with and resulted in the same shaped tree, but the methodology used was different but their amortised cost were the same.*

| | | | |
|---|---|---|---|
| Count of nodes | 11 | | |
| Start potential | 18 | End potential | 15 |
| Comparisons | 0 | Amortised | 3 |

Case 12: Refer to Figure 2.19.

Removing any one of the three black nodes will bring the new shape within standard arrangement on the right. The potential is decreased by three with no comparisons spent.

| | | | |
|---|---|---|---|
| Count of nodes | 10 | | |
| Start potential | 15 | End potential | 12 |
| Comparisons | 0 | Amortised | 3 |

Case 13: Refer to Figure 2.20.

Removing any one of the four black nodes will cause the new shape to

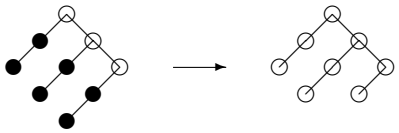

Figure 2.21: Decrease-key case 14

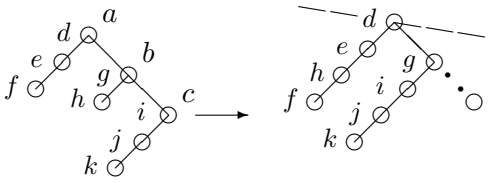


Figure 2.22: Decrease-key case 14 - heap transformation

violate the standard arrangement rules. This is resolved by taking advantage of the non-increasing order rule of the nodes. The same restructuring process can be used regardless of which one of the four black nodes was removed. In this particular example, node $(i)$ was removed and to restructure the following was done: remove nodes $(d)$, $(e)$ and $(f)$ from trunk $a$, delete trunk $c$ by removing nodes $(g)$ and $(h)$ and insert them beneath trunk $a$ using no comparisons. Using one comparison, compare node $(d)$ with the $(i+1)$-th dimension node of trunk $b$, and create a new trunk in the correct $(i+1)$-th dimension position using nodes $(d)$, $(e)$ and $(f)$. In this example the value of node $(d)$ was larger than the $(i+1)$-th dimension node of trunk $b$, so trunk $b$ was not displaced by the new trunk '$*$'. The new standard arrangement is on the right. The potential is decreased by three with one comparison spent.

*Note: The previous case started with and resulted in the same shaped tree, but the methodology used and its amortised cost are different.*

| | | | |
|---|---|---|---|
| Count of nodes | 10 | | |
| Start potential | 15 | End potential | 12 |
| Comparisons | 1 | Amortised | 4 |

Case 14: Refer to Figures 2.21–2.23.

Removing any one of the six black nodes from the left tree in Figure 2.21, will cause the new shape to violate the standard arrangement rules with the new temporary shape on the right. Since this tree has less than the

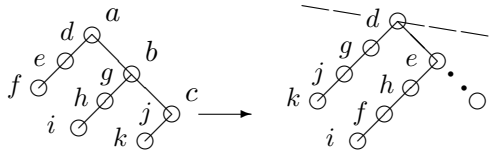


Figure 2.23: Decrease-key case 14 - another heap transformation

minimum required nine nodes it cannot be restructured back into standard arrangement without the make-up process.

The make-up process occurs in the workspaces defined by the $(i + 1)$-th dimension (dotted line) and the $(i + 2)$-th dimension (dashed line) as highlighted in Figure 2.22. This process may repeat many times and will stop in one of cases 1–13, or until no higher trunk exists. A critical property required for moving this from one dimension into another is for the amortised cost to equal zero.

The right tree shown in Figure 2.22 and 2.23, represents the final transformation shape of this tree when it is just about ready to be moved into the higher dimension. It is this tree which will be used to calculate the end potential for this scenario. Please note that the node which got removed is shown in this tree by the dotted line and has been left in for accounting purposes.

The process to resolve this tree into a shape ready for the make-up process depends on which trunk is violating standard arrangement and these are handled as follows:

Trunks $a$ and $b$:

The left tree in Figure 2.22 is the temporary tree from Figure 2.21 and in this particular example trunk $b$ lost the node. Restructuring is achieved as follows: on trunk $b$ remove node $(h)$ and insert this into trunk $a$ using two comparisons, see Section 2.7 comparison algorithm $B$. Next delete trunk $c$ by removing nodes $(i)$, $(j)$, and $(k)$, and insert these into trunk $b$ using no comparisons. The new tree shape ready for the make-up process is on the right, a total of two comparisons were used.

Trunk $c$:

The left tree in Figure 2.23 is the temporary tree from Figure 2.21 and

restructuring is achieved as follows: remove four consecutive nodes $(d)$, $(g)$, $(j)$ and $(k)$, and construct a full sized trunk using no comparisons. Next merge together the two orphaned two-node trunks, $(e,\ f)$ and $(h,\ i)$. This merge operation can be achieved using three comparisons as detailed in Section 2.7 comparison algorithm $A$. No comparisons are required to add this trunk with the other because each node $(e,\ f,\ h,\ i)$ is known to be larger than node $(d)$ and can therefore can be simply inserted into position. The new tree shape ready for the make-up process is on the right, a total of three comparisons were used.

Even though trunks $a$ and $b$ required two comparisons, the trunk $c$ required three. It is the most costly comparison cost which is recorded.

Studying the below table, it can be seen that in the worst case situation the amortised cost of this transformation into a higher dimension is zero. The critical amortised cost equalling zero has been achieved. Of great interest however is if the node removed occurred on either trunks $a$ or $b$, then the amortised cost would be negative one and this scenario would have made a profit.

| | | | |
|---|---|---|---|
| Count of nodes | 9 | | |
| Start potential | 12 | End potential | 15 |
| Comparisons | 3 | Amortised | 0 |

### *2.8.2 Top Level Insertions*

Suppose we are going to insert a tree of type $T(i)$ into the top level tree where the top level tree may already contain a tree of main trunk $\mathbf{a_i}$. There are four cases:

Case A. $\mathbf{a_i = 0}$: No nodes, simply put the tree into the correct position.

Case B. $\mathbf{a_i = 1}$: One existing node, can form a new $\mathbf{2}T(i)$ with one comparison, and increase the potential by one.

Case C. $\mathbf{a_i = 2}$: Two existing nodes, can form a new $\mathbf{3}T(i)$ with two comparisons, and increase the potential by two.

Case D. $\mathbf{a_i = 3}$: Three existing nodes, can make a new $\mathbf{4}T(i)$ with three comparisons, worst case, and increase the potential by three. Because this tree has reached its maximum size for this top level position, we force a carry and do an insertion at $\mathbf{a_{i+1}}T(i+1)$.

### *2.8.3 Insert*

Insertion is covered by the above four cases of A, B, C, and D where insertion of type $T(0)$ tree, a tree with only a single node.

### *2.8.4 Delete-min*

A delete-min operation removes the minimum key valued node from a tree and this node is also referred to as the root node. To locate the smallest key valued node in the heap, linear search is performed starting from the highest tree $\mathbf{a_{k-1}}T(k-1)$ down to $\mathbf{a_0}T(0)$ and key comparisons are incurred during this process. After the deletion of the root node from its tree, $\mathbf{a_i}T(i)$, the tree will have been broken apart into smaller sub-trees $\mathbf{b_0}T(0), \ \ldots, \ \mathbf{b_i}T(i)$, where each $\mathbf{b_j}$ is **2**, or **3** for top level positions $j = 0, \ \ldots, \ i-1$, where i represents the current tree's top level position. When $\mathbf{a_i}$ is **1**, **2**, or **3**, then $\mathbf{b_i} = (\mathbf{a_i} - \mathbf{1})$. If $\mathbf{b_i} = 0$ then there are will be no trees left over in the current position $T(i)$.

Looking at Figure 2.6 diagram $D$, $\mathbf{a_i} = \mathbf{3}$ because it is constructed of three degree two (2) heaps. After breaking up this tree, $\mathbf{b_i} = \mathbf{2}$ because the left-most tree was broken up into sub-trees $\mathbf{b_j}$ $(j = 0, \ \ldots, \ i-1)$, and therefore, the right-most trees will make $\mathbf{b_i} = \mathbf{2}$. The two trees represented by $\mathbf{b_i}$ shall remain behind in the top level position $T(i)$. Next the sub-trees represented by $\mathbf{b_j}$ are merged into any existing trees in top level positions $j = 0, \ \ldots, \ i-1$. In general trees $\mathbf{a_j}T(j)$, $\mathbf{b_j}T(j)$ and $\mathbf{c_j}T(j)$ are merged. Term $\mathbf{a_j}T(j)$ is used to represent any existing tree in top level position $T(j)$. Term $\mathbf{c_j}T(j)$ is used to represent if there is a carry happening from the $(j-1)$-th position. If $\mathbf{c_j} = \mathbf{1}$ then there is a carry to handle from a lower top level position, and if $\mathbf{c_j} = \mathbf{0}$ then there is no carry. Term $\mathbf{b_j}T(j)$ represents one fragment of the tree which was broken up by the removal of the minimum node.

The remainder of this section will analyse the amortised cost for the thirty two (32) combinations of ($\mathbf{a_j}$, $\mathbf{b_j}$, $\mathbf{c_j}$). Table. 2.2 contains a description of the legend used by the following tables:

**Case**$(\mathbf{0}, \mathbf{0}, \mathbf{0})$

Nothing occurs in this scenario because $\mathbf{a_j}$, $\mathbf{b_j}$, and $\mathbf{c_j}$ are empty.

| Legend | Description |
|---|---|
| Starting potential | Represents the pre-merging combined potential of $\mathbf{a_j}$, $\mathbf{b_j}$ and $\mathbf{c_j}$ |
| Comparisons | Represents number of comparisons required to complete the merging of $\mathbf{a_j}$, $\mathbf{b_j}$ and $\mathbf{c_j}$ |
| End potential | Represents the post-merging combined potential of all trees which are used to make up the new $\mathbf{a_j}T(j)$ and carry $\mathbf{c_{j+1}}$T(j + 1) |
| Amortised | Merge operation total amortised cost. This is calculated by adding the starting potential and comparisons together and then subtracting the end potential. The expected amortised cost should be zero. |

Table 2.2: Terms used during amortised cost analysis

| | | | |
|---|---|---|---|
| Starting potential | 0 | Comparisons | 0 |
| End potential | 0 | Amortised | 0 |

**Case**$(\mathbf{0}, \mathbf{0}, \mathbf{1})$

There is a carry $\mathbf{c_j}$ from a lower top level position, $\mathbf{a_j}$ and $\mathbf{b_j}$ are empty. Insert $\mathbf{c_j}$ using Case A.

| | | | |
|---|---|---|---|
| Starting potential | 0 | Comparisons | 0 |
| End potential | 0 | Amortised | 0 |

**Case**$(\mathbf{0}, \mathbf{1}, \mathbf{0})$

There is no tree in $\mathbf{a_j}$ or a carry in $\mathbf{c_j}$, insert $\mathbf{b_j}$ using Case A.

| | | | |
|---|---|---|---|
| Starting potential | 0 | Comparisons | 0 |
| End potential | 0 | Amortised | 0 |

**Case**$(\mathbf{0}, \mathbf{1}, \mathbf{1})$

There is no tree in $\mathbf{a_j}$, insert tree $\mathbf{b_j}$ using Case A and then merge in $\mathbf{c_j}$ using Case B.

| | | | |
|---|---|---|---|
| Starting potential | 0 | Comparisons | 1 |
| End potential | 1 | Amortised | 0 |

**Case**$(\mathbf{0}, \mathbf{2}, \mathbf{0})$

There is no tree in $\mathbf{a_j}$ or a carry in $\mathbf{c_j}$, insert $\mathbf{b_j}$ using Case A.

| | | | |
|---|---|---|---|
| Starting potential | 1 | Comparisons | 0 |
| End potential | 1 | Amortised | 0 |

**Case**$(\mathbf{0}, \mathbf{2}, \mathbf{1})$

There is no tree in $\mathbf{a_j}$, insert tree $\mathbf{b_j}$ using Case A and merge in $\mathbf{c_j}$ using Case C.

| Starting potential | 1 | Comparisons | 2 |
|---|---|---|---|
| End potential | 3 | Amortised | 0 |

**Case**$(\mathbf{0}, \mathbf{3}, \mathbf{0})$

There is no tree in $\mathbf{a_j}$ or a carry in $\mathbf{c_j}$, insert $\mathbf{b_j}$ using Case A.

| Starting potential | 3 | Comparisons | 0 |
|---|---|---|---|
| End potential | 3 | Amortised | 0 |

**Case**$(\mathbf{0}, \mathbf{3}, \mathbf{1})$

There is no tree in $\mathbf{a_j}$, insert tree $\mathbf{b_j}$ into position and then merge in $\mathbf{c_j}$ using Case D.

Before merging the combined potential of $\mathbf{b_j}$ and $\mathbf{c_j}$ is measured as being three. Two comparisons were used to merge these two trees, resulting in a combined potential of six. See Section 2.7 algorithm $B$ on how to merge three node and single node trunk together. The amortised cost of this operation is minus one, a profit has been made. See Section 2.6 for detailed analysis of negative amortised cost.

With the tree at its maximum size of four, this will cause it to be carried into the next higher top level position, resulting in the current top-level position $T(j)$ becoming empty. The carry operation is not handled by this scenario but that prevalent for the top-level position $T(j+1)$.

| Starting potential | 3 | Comparisons | 2 |
|---|---|---|---|
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{1}, \mathbf{0}, \mathbf{0})$

There is no tree in $\mathbf{b_j}$ or $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. There is nothing to do in this case.

| Starting potential | 0 | Comparisons | 0 |
|---|---|---|---|
| End potential | 0 | Amortised | 0 |

**Case**$(\mathbf{1}, \mathbf{0}, \mathbf{1})$

There is no tree in $\mathbf{b_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in $\mathbf{c_j}$ using Case B.

| Starting potential | 0 | Comparisons | 1 |
|---|---|---|---|
| End potential | 1 | Amortised | 0 |

**Case**$(\mathbf{1}, \mathbf{1}, \mathbf{0})$

There is no tree in $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in $\mathbf{b_j}$ using

Case B.

| | | | |
|---|---|---|---|
| Starting potential | 0 | Comparisons | 1 |
| End potential | 1 | Amortised | 0 |

**Case**$(\mathbf{1}, \mathbf{1}, \mathbf{1})$

Tree $\mathbf{a_j}$ is already in position so merge in $\mathbf{b_j}$ using Case B and then $\mathbf{c_j}$ using Case C.

| | | | |
|---|---|---|---|
| Starting potential | 0 | Comparisons | 3 |
| End potential | 3 | Amortised | 0 |

**Case**$(\mathbf{1}, \mathbf{2}, \mathbf{0})$

There is no tree in $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in $\mathbf{b_j}$ using Case C.

| | | | |
|---|---|---|---|
| Starting potential | 1 | Comparisons | 2 |
| End potential | 3 | Amortised | 0 |

**Case**$(\mathbf{1}, \mathbf{2}, \mathbf{1})$

Tree $\mathbf{a_j}$ is already in position, merge in $\mathbf{b_j}$ using Case C, and then merge in $\mathbf{c_j}$ using Case D. Before merging the combined potential of $\mathbf{a_j}$, $\mathbf{b_j}$, and $\mathbf{c_j}$ was one. Four comparisons were used to merge these three trees and resulted in a combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Alternatively the merge pattern used could have been to merge $\mathbf{a_j}$ and $\mathbf{c_j}$ using Case B, and then merge in $\mathbf{b_j}$. Four comparisons were also used by the alternative approach. Because the tree has reached maximum size, this will cause a carry leaving top-level position $T(j)$ empty.

| | | | |
|---|---|---|---|
| Starting potential | 1 | Comparisons | 4 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{1}, \mathbf{3}, \mathbf{0})$

There is no tree in $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in $\mathbf{b_j}$ using Case D. Before merging the combined potential was three. Two comparisons where used to merge these two trees, resulting in a combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Because the tree has reached maximum size, this will cause a carry leaving top-level position $T(j)$ empty.

| | | | |
|---|---|---|---|
| Starting potential | 3 | Comparisons | 2 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{1}, \mathbf{3}, \mathbf{1})$

With tree $\mathbf{a_j}$ already in position, merge in $\mathbf{b_j}$ using Case D. Before merging the combined potential was three. Two comparisons where used to merge these two trees, resulting in a combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Because the tree has reached maximum size, this will cause a carry leaving top-level position $T(j)$ empty. Using Case A, insert tree $\mathbf{c_j}$ into $T(j)$.

| | | | |
|---|---|---|---|
| Starting potential | 3 | Comparisons | 2 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{2}, \mathbf{0}, \mathbf{0})$

There is no tree in $\mathbf{b_j}$ or $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. There is nothing to do in this case.

| | | | |
|---|---|---|---|
| Starting potential | 1 | Comparisons | 0 |
| End potential | 1 | Amortised | 0 |

**Case**$(\mathbf{2}, \mathbf{0}, \mathbf{1})$

There is no tree in $\mathbf{b_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in $\mathbf{c_j}$ using Case C.

| | | | |
|---|---|---|---|
| Starting potential | 1 | Comparisons | 2 |
| End potential | 3 | Amortised | 0 |

**Case**$(\mathbf{2}, \mathbf{1}, \mathbf{0})$

There is no tree in $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in $\mathbf{b_j}$ using Case C.

| | | | |
|---|---|---|---|
| Starting potential | 1 | Comparisons | 2 |
| End potential | 3 | Amortised | 0 |

**Case**$(\mathbf{2}, \mathbf{1}, \mathbf{1})$

Tree $\mathbf{a_j}$ is already in position. Merge in tree $\mathbf{b_j}$ using Case C and then merge in $\mathbf{c_j}$ using Case D. Before merging the combined potential was one. Four comparisons where used to merge these three trees, resulting in a combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Because the tree has reached maximum size, this will cause a carry leaving top-level position $T(j)$ empty.

| | | | |
|---|---|---|---|
| Starting potential | 1 | Comparisons | 4 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{2}, \mathbf{2}, \mathbf{0})$

There is no tree in $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in $\mathbf{b_j}$ using three comparisons. See to Section 2.7 algorithm $A$ on how to merge

two two-node trunks together. Before merging the combined potential was two. Three comparisons where used to merge these two trees, resulting in a combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Because this tree has reached maximum size, this will cause a carry leaving top-level position $T(j)$ empty.

| | | | |
|---|---|---|---|
| Starting potential | 2 | Comparisons | 3 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{2}, \mathbf{2}, \mathbf{1})$

Tree $\mathbf{a_j}$ is already in position. Merge in $\mathbf{b_j}$ using three comparisons. Before merging the combined potential was two. Three comparisons where used to merge these two trees, resulting in a combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Because this tree has reached maximum size, this will cause a carry leaving top-level position $T(j)$ empty. Using Case A, insert tree $\mathbf{c_j}$ into $T(j)$.

| | | | |
|---|---|---|---|
| Starting potential | 2 | Comparisons | 3 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{2}, \mathbf{3}, \mathbf{0})$

There is no tree in position $\mathbf{c_j}$. Remove the top-most (smallest) node from $\mathbf{a_j}$ and merge this into $\mathbf{b_j}$. Before merging and node removal the combined potential was four. With the removal of a node from tree $\mathbf{a_j}$, its potential was reduced from one to zero. Two comparisons were used to merge in the top-most node from $\mathbf{a_j}$ into $\mathbf{b_j}$, bring a new combined potential of six. The amortised cost of this operation is zero and because the tree has reached maximum size, a carry is caused. At the end of this operation, tree $\mathbf{a_j}$ has only one node.

| | | | |
|---|---|---|---|
| Starting potential | 4 | Comparisons | 2 |
| End potential | 6 | Amortised | 0 |

**Case**$(\mathbf{2}, \mathbf{3}, \mathbf{1})$

There is a tree in position $\mathbf{a_j}$ and this is left untouched. Merge together trees $\mathbf{b_j}$ and $\mathbf{c_j}$. Before merging the combined potential of all three trees was four. Two comparisons were used to merge together $\mathbf{b_j}$ and $\mathbf{c_j}$ causing them to reach maximum size and carry. With the end potential being seven, the amortised cost of this operation is minus one, a profit has been made.

| | | | |
|---|---|---|---|
| Starting potential | 4 | Comparisons | 2 |
| End potential | 7 | Amortised | -1 |

**Case**$(\mathbf{3}, \mathbf{0}, \mathbf{0})$

There is no tree in $\mathbf{b_j}$ or $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. There is nothing to do in this case.

| | | | |
|---|---|---|---|
| Starting potential | 3 | Comparisons | 0 |
| End potential | 3 | Amortised | 0 |

**Case**$(\mathbf{3}, \mathbf{0}, \mathbf{1})$

There is no tree in $\mathbf{b_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in tree $\mathbf{c_j}$ using Case D. Before merging the combined potential was three and two comparisons were used to merge these two trees, resulting in a new combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Because the tree has reached maximum size, a carry is caused leaving top-level position $T(j)$ empty.

| | | | |
|---|---|---|---|
| Starting potential | 3 | Comparisons | 2 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{3}, \mathbf{1}, \mathbf{0})$

There is no tree in $\mathbf{c_j}$ and tree $\mathbf{a_j}$ is already in position. Merge in tree $\mathbf{b_j}$ using Case D. Before merging the combined potential was three and two comparisons were used to merge these two trees, resulting in a new combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Because the tree has reached maximum size, a carry is caused leaving top-level position $T(j)$ empty.

| | | | |
|---|---|---|---|
| Starting potential | 3 | Comparisons | 2 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{3}, \mathbf{1}, \mathbf{1})$

Tree $\mathbf{a_j}$ is already in position, merge in tree $\mathbf{b_j}$. Before merging the combined potential was three and two comparisons were used to merge these two trees, resulting in a new combined potential of six. The amortised cost of this operation is minus one, a profit has been made. Because the tree has reached maximum size, a carry is caused. Using Case A, insert tree $\mathbf{c_j}$ into $T(j)$.

| | | | |
|---|---|---|---|
| Starting potential | 3 | Comparisons | 2 |
| End potential | 6 | Amortised | -1 |

**Case**$(\mathbf{3}, \mathbf{2}, \mathbf{0})$

There is no tree in position $\mathbf{c_j}$. Remove the top-most node from $\mathbf{b_j}$ and merge this into $\mathbf{a_j}$. Before merging and node removal the combined potential was four. With the removal of a node from tree $\mathbf{b_j}$, its potential reduces from one to zero. Two comparisons were used to merge in the top-most node from $\mathbf{b_j}$ into $\mathbf{a_j}$ resulting in a new combined potential of six. The amortised cost of this operation is zero and because the tree has reached maximum size, this will cause a carry. Using Case A, insert the remainder of tree $\mathbf{b_j}$ into $T(j)$.

| | | | |
|---|---|---|---|
| Starting potential | 4 | Comparisons | 2 |
| End potential | 6 | Amortised | 0 |

**Case**$(\mathbf{3}, \mathbf{2}, \mathbf{1})$

There is a tree in position $\mathbf{a_j}$. Merge together trees $\mathbf{a_j}$ and $\mathbf{c_j}$. Before merging the combined potential of all three trees was four. Two comparisons were used to merge together $\mathbf{a_j}$ and $\mathbf{c_j}$ causing them to reach maximum size and carry. With the end potential being seven, the amortised cost of this operation is minus one, a profit has been made. Using Case A, insert tree $\mathbf{b_j}$ into $T(j)$.

| | | | |
|---|---|---|---|
| Starting potential | 4 | Comparisons | 2 |
| End potential | 7 | Amortised | -1 |

**Case**$(\mathbf{3}, \mathbf{3}, \mathbf{0})$

Compare the top-most nodes of $\mathbf{a_j}$ and $\mathbf{b_j}$. Remove the smaller of these two nodes and insert it into the other trunk by making it the new top-most node. One comparison is used and the amortised cost is zero. Because this tree has reached maximum size, a carry is caused. If the top-most node was removed from tree $\mathbf{b_j}$, then insert the remaining two nodes of tree $\mathbf{b_j}$ into the empty $T(j)$ using Case A.

| | | | |
|---|---|---|---|
| Starting potential | 6 | Comparisons | 1 |
| End potential | 7 | Amortised | 0 |

**Case**$(\mathbf{3}, \mathbf{3}, \mathbf{1})$

There is a tree in position $\mathbf{a_j}$ and this is left untouched. Merge together trees $\mathbf{b_j}$ and $\mathbf{c_j}$. Before merging the combined potential of all three trees was six. Two comparisons were used to merge together $\mathbf{b_j}$ and $\mathbf{c_j}$ causing them to reach maximum size and carry. With the end potential being nine, the amortised cost of this operation is minus one, a profit has been made.

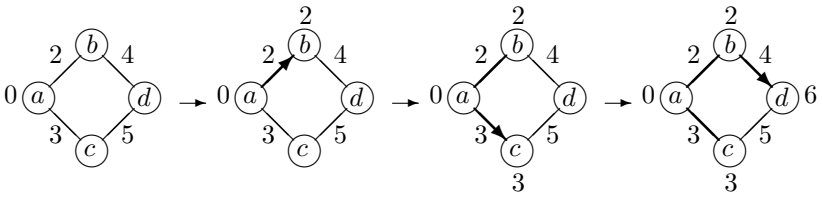


Figure 2.24: Dijkstra example

| Starting potential | 6 | Comparisons | 2 |
|---|---|---|---|
| End potential | 9 | Amortised | -1 |

### *2.9 Dijkstra Algorithm*

Dutch computer scientist Edsger Dijkstra [2] in 1959 invented a graph search algorithm which solves the single source shortest path problem. This algorithm will determine the shortest path, also known as lowest cost, between any given vertex and all other vertices in the graph. Each vertex has got a label associated with it and this represents the calculated distance cost from the source vertex to that particular vertex. This label is unassigned if the distance has not been calculated. Starting from the start vertex, it will repeatedly examine the closest not-yet-examined vertex and expands outwards until all vertices are labelled. This algorithm is formally expressed in Figure 2.25. Dijkstra algorithm complexity is expressed as $O(m+n\log n)$, where $m$ is the total number of edges and $n$ is the total number of vertices. Once $n$ insert, $n$ delete-min and $m$ decrease-key operations have been performed by Dijkstra algorithm, the end potential is zero.

Suppose you have a [9] "knotted web of strings, with each knot corresponding to a node, and the strings corresponding to the edges of the web: the length of each string is proportional to the weight of each edge." Now lay this web on the floor and select your starting knot. Slowly lift the web off the floor and as each knot leaves the surface, its label can be calculated as being the total distance from the starting knot to itself.

Figure 2.24 represents a sample web which goes through process of being lifted off the floor, starting from the left. The starting vertex is $a$ and its label

value is zero and all other labels are left unassigned. The edge which lifts the vertex off the floor is highlighted with an arrow. As the web is lifted off the floor the first vertex to be lifted is vertex $b$. Its label value is calculated as being the edge cost plus vertex $a$ value, $2 + 0 = 2$. Continue lifting and the next vertex to lift off the floor is vertex $c$. Its label value is calculated as being the edge cost plus vertex $a$ value, $3 + 0 = 3$. Continue lifting and the final vertex to lift off the floor is vertex $d$. Its label value is calculated as being the edge cost plus vertex $b$ value, $4 + 2 = 6$.

```
1.  S := φ;
2.  Put s into S; d[s]:=0;
3.  for v in OUT(s) do d[v]:= L(s, v); { L(u, v) is the length of edge (u,
    v) }
4.  F := { v | (s, v) is in E };
5.  while F is not empty do begin
6.     v := u such that d[u] is minimum among u in F
7.     F := F - {v}; S := S ∪ {v};
8.     for w in OUT(v) do
9.        if w is not in S then
10.          if w is in F then d[w] := min { d[w], d[v]+L(v, w) }
11.          else begin d[w] := d[v]+L(v, w); F := F ∪ {w} end
12. end
```

Figure 2.25: Dijkstra algorithm pseudo code [15]

Figure 2.25 represents the pseudo code of Dijkstra's 'Single Source Shortest Path' algorithm. Next is a brief explanation of the pseudo code symbols followed by line analysis.

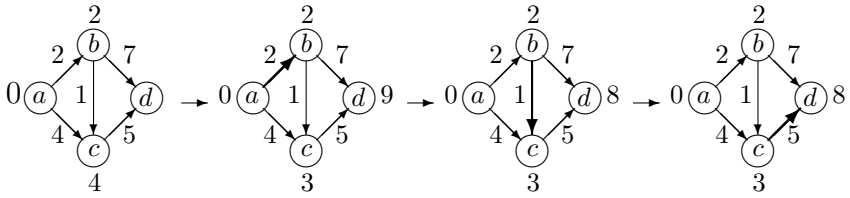


Figure 2.26: Dijkstra example with decrease-key

Symbol $S$ is the set of vertices to which the shortest distances are finalised. Symbol $F$ is the set of vertices to which there are direct connections from vertex $s$ and this is the set to be organised into the heap. Symbol '$OUT(s)$' is the set of vertices to which there are direct connections from $S$.

Line 1: Defines the solution set as symbol $S$ and it is empty to start with.

Line 2: Add the source vertex $s$ into solution set $S$. Set its label to zero.

Line 3: Iterate over set of vertices represented by $v$ in '$OUT(s)$' and assign labels to those vertices.

Line 4: Initialisation of frontier set $F$.

Line 5: Start iteration loop while the frontier set $F$ is not empty.

Line 6: The smallest key valued vertex in frontier set $F$ has the delete-min operation performed on it. In relation to Figure 2.24, "delete-min" can be described as holding vertex $a$ in your hand and cutting the strings which represent the connecting edges with vertices $b$ and $c$.

Line 7: Remove vertex $v$ from frontier set $F$ and add into solution set $S$.

Line 8: Iterate over vertices set in $OUT(v)$. Each vertex is represented by $w$.

Line 9: If vertex $w$ is not in solution set $S$ then continue to line 10.

Line 10: If vertex $w$ is in frontier set $F$ then perform "decrease-key" on it.

Line 11: If vertex $w$ is not in frontier set $F$ but can be visited from '$OUT(v)$' then assign label to vertex $w$ and "insert" it into frontier set $F$.

Line 12: Marks end of the iterating loop started at line 5.

Figure 2.26 represents a sample web which goes through process of being lifted off the floor, starting from the left. The difference with this sample web and that of Figure 2.24 is an extra connecting line between vertices $b$ and $c$, directional edges, and the inclusion of the decrease-key process. The

edge which lifts the vertex off the floor is highlighted by becoming thicker.

The source vertex is $a$, its label value is zero and it is added into solution set $S$. All other vertices are left unassigned except for vertices $b$ and $c$ which have a directional edge connecting them with vertex $a$ in solution set $S$. Vertices $b$ and $c$ are added into frontier set '$F$' and their label values are respectively set to the value of the connecting edge plus vertex $a$ value.

As the web is lifted off the floor the first vertex to be lifted is vertex $b$. Vertex $c$ is in the set of '$OUT(b)$' and is included in the frontier set $F$, recalculate its label value as being the edge cost plus vertex $b$ value, $1+2 = 3$, and then perform a decrease-key on its label value. Vertex $d$ is in the vertices set of '$OUT(b)$' and not in the frontier set $F$, its label value is calculated as being the edge cost plus vertex $b$ value, $7 + 2 = 9$, and it is added into the frontier set $F$. Vertex $b$ has now been visited so its distance is finalised. Remove vertex $b$ from the frontier set $F$ and add into solution set $S$.

Continue lifting the web and the next vertex to lift of the floor is vertex $c$. Vertex $d$ is in the set of '$OUT(c)$' and is included in the frontier set $F$, recalculate its label value as being the edge cost plus vertex $c$ value, $3+5 = 8$, and then perform a decrease-key on its label value. Vertex $c$ has now been visited so its distance is finalised. Remove vertex $c$ from the frontier set $F$ and add into solution set $S$.

Continue lifting the web and the final vertex to lift off the floor is vertex $d$. There are no vertices to visit in the set of '$OUT(d)$'. Remove vertex $d$ from the frontier set $F$ and add into solution set $S$.

# Chapter III

# 3-4 Heap Experiments

## *3.1 Experiment Setup*

These experiments investigate the unique features of the 3-4 heap as an alternative to the 2-3 heap and concentrate on the core workspace operations of insert, delete-min and decrease-key.

For each experiment, the number of points in the data set graph range from one hundred (100) to five thousand (5,000). Graph sizes between one hundred (100) to five hundred (500) points used an incremental size of fifty (50), and graph sizes between one thousand (1000) to five thousand (5000) points used an incremental size of five hundred (500). The letter '$n$' is used to represent the total number of points contained in each dataset during an experiment.

One threat to experiment validity is having identical experiments being performed on each data structure using a slightly different graph data set. To ensure that this does not occur, both data structures will be operated in series with the same graph data set.

## *3.2 Insert Experiments*

This section details the number of node-to-node key comparisons required by both the 2-3 heap and 3-4 heap during insert operations.

Different insertion experiments were performed where the values inserted were a) incremented by one (1), b) decremented by one (1), c) randomly generated using modulus of five hundred (500), and d) using a series of four numerical values where the series seed is randomly generated using modulus of five hundred (500) followed by three increments of one (1).

Experiment results data used to generate each figure presented in this section are shown in Table 3.1 on page 45.

### *3.2.1 Increasing Numerical*

Inserting increasing numerical data will generate the largest number of key comparisons for node insertion because each node is inserted at the end of a trunk and since both heaps have very similar operating characteristics, their respective key comparison cost increase at a constant rate. However the 3-4 heap required more key comparisons than the 2-3 heap.

The reason behind these additional key comparisons is driven by having a fourth node on the trunk and the number of comparisons required inserting the fourth node into its proper position. The insertion method used does not use linear comparison scanning because if it did, there would be up to three comparisons used and thus the 3-4 heap would have used more key comparisons than it did. The actual method harnessed uses a constant rate of two key comparisons. This is achieved by compared first the middle node on the three node trunk and then one at either end of the trunk depending on the first comparison outcome. The second comparison will determine where on the trunk the fourth node will be inserted. This gives the 3-4 heap a constant cost of two comparisons versus a floating comparison cost of between one to three comparisons. With this type of numerical data, this makes the insertion of the fourth node less expensive by one comparison than achieved by using linear comparison scanning. In comparison, the 2-3 heap utilises linear search to locate a node's insertion point.

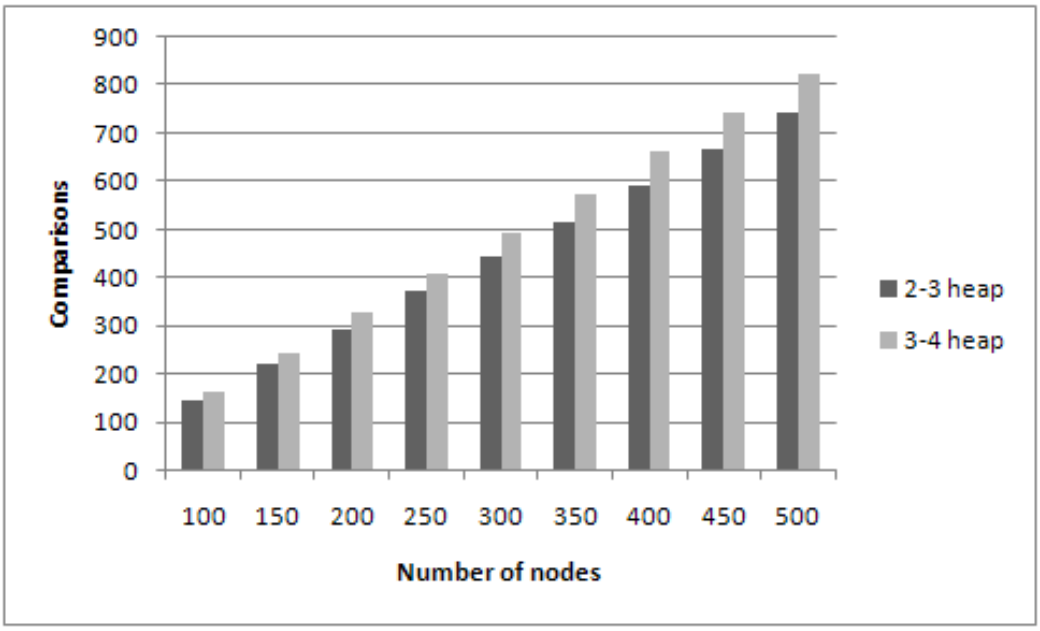


Figure 3.1: Insert increasing numerical value using standard 2-3 heap and 3-4 heap

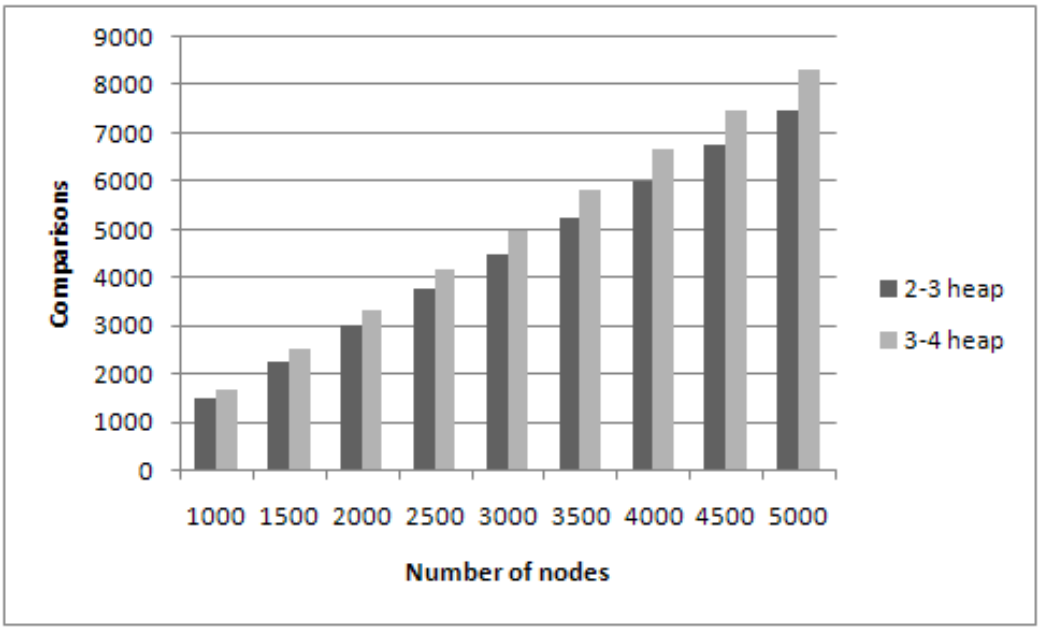


Figure 3.2: Insert increasing numerical value using standard 2-3 heap and 3-4 heap

### *3.2.2 Decreasing Numerical*

Inserting decreasing numerical data will generate the smallest number of key comparisons for node insertion because each node is inserted at the front of a trunk and since both heaps have very similar operating characteristics, their respective key comparison cost increased at a constant rate. However the 3-4 heap required more key comparisons than the 2-3 heap. As noted in Section 3.2.1, the 3-4 heap has a flat cost of two comparisons to determine trunk insertion point of the fourth node and because of this fixed comparison cost, this makes the insertion of the fourth node more expensive by one comparison than the most efficient possible cost achieved by linear comparison scanning.

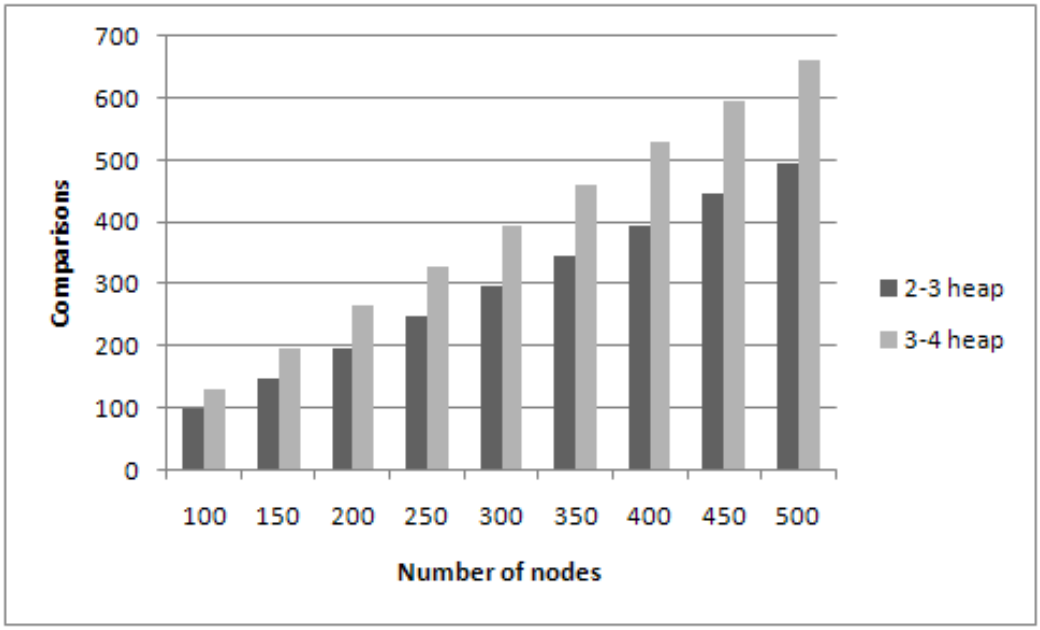


Figure 3.3: Insert decreasing numerical value using standard 2-3 heap and 3-4 heap

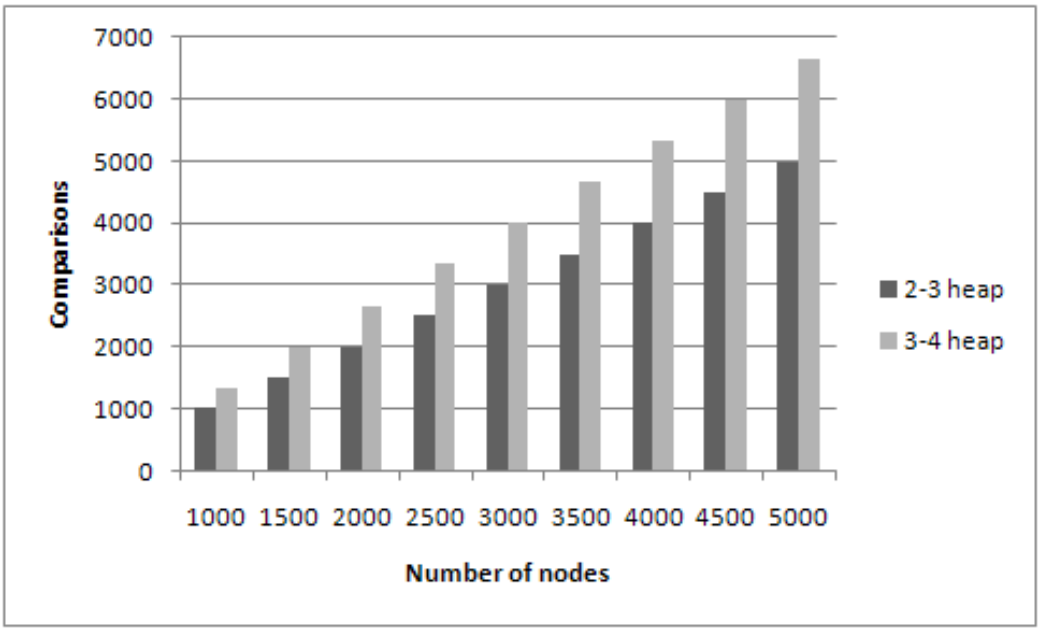


Figure 3.4: Insert decreasing numerical value using standard 2-3 heap and 3-4 heap

### *3.2.3 Random Numerical*

Inserting random numerical data will generate key comparisons based upon the inserting of numerical values occurring in a mixture of positions on the trunk and since both heaps have very similar operating characteristics, their respective key comparison cost increased at a constant rate. However the 3-4 heap required more key comparisons than the 2-3 heap as noted in Section 3.2.1.

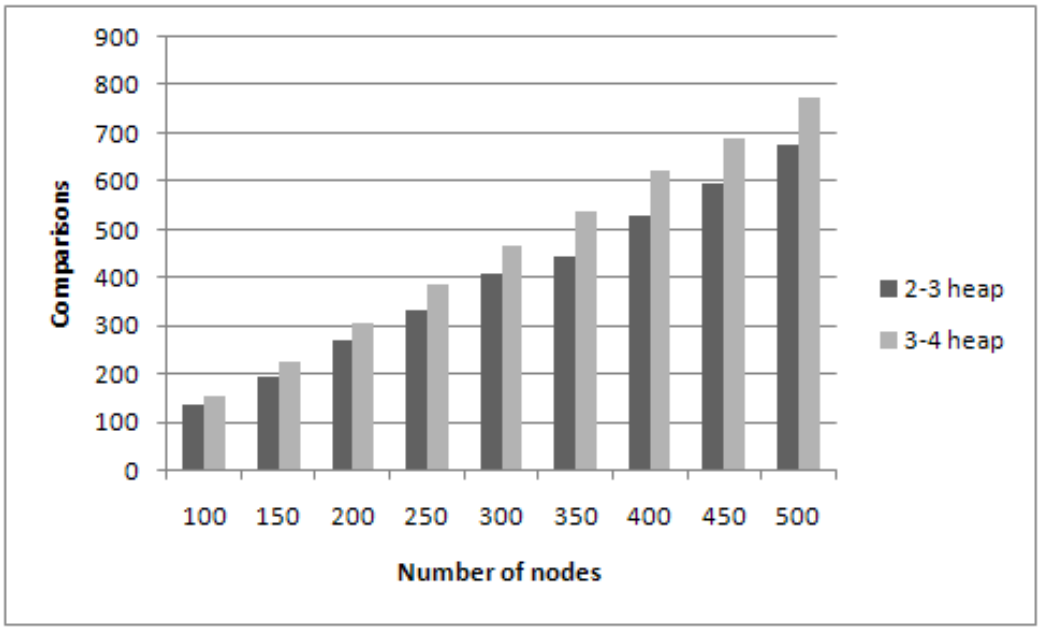


Figure 3.5: Insert randomly generated modulus 500 numerical value using standard 2-3 heap and 3-4 heap

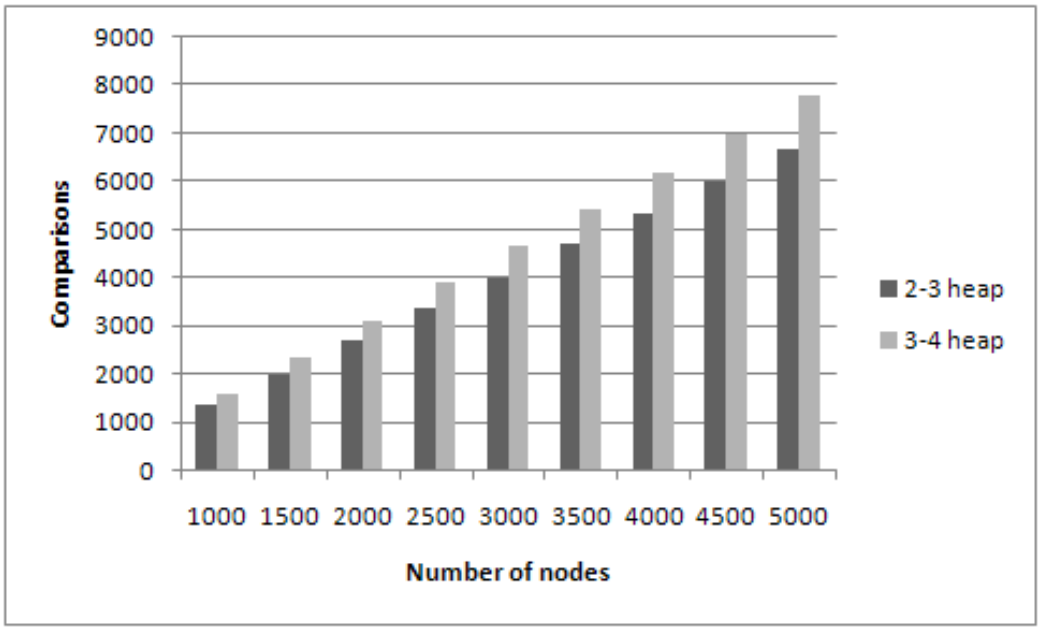


Figure 3.6: Insert randomly generated modulus 500 numerical value using standard 2-3 heap and 3-4 heap

### *3.2.4 Osculating Numerical*

Inserting osculating numerical data will generate key comparisons based upon the insertion of a series being incremented by one (1) consecutively three times before a new series seed was randomly generated with modulus five hundred (500). This ensures there are ascending runs of length 4 all the way in the input sequence.

Since both heaps have very similar operating characteristics, their respective key comparison cost increased at a constant rate. However the 3-4 heap required more key comparisons than the 2-3 heap as noted in Section 3.2.1.

While designing this experiment it was thought that this type of input data might display a difference in key comparisons incurred by the 3-4 heap because the series was more aligned towards the 3-4 heap trunk size, but this wasn't the case. The results displayed an inconclusive difference. There will be more conclusive difference with some heap modification in the next section.

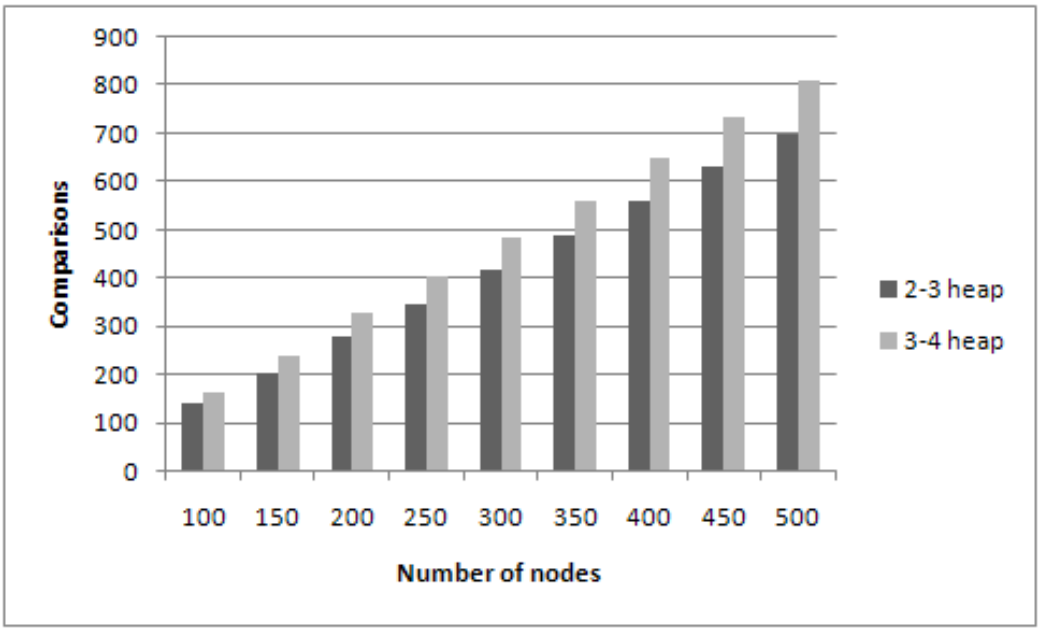


Figure 3.7: Insert osculating generated modulus 500 numerical value using standard 2-3 heap and 3-4 heap

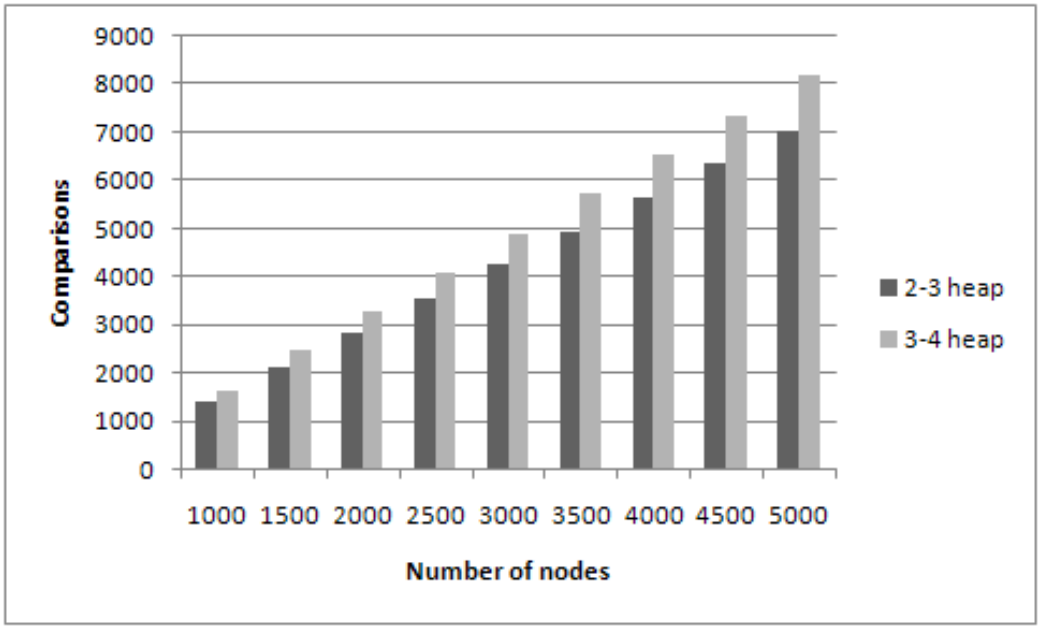


Figure 3.8: Insert osculating generated modulus 500 numerical value using standard 2-3 heap and 3-4 heap

| Number | Increasing | | Decreasing | | Random | | Osculating | |
|---|---|---|---|---|---|---|---|---|
| of Nodes | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap |
| 100 | 145 | 161 | 97 | 129 | 135 | 150 | 138 | 159 |
| 150 | 218 | 242 | 146 | 194 | 192 | 222 | 203 | 238 |
| 200 | 293 | 329 | 196 | 263 | 268 | 306 | 276 | 325 |
| 250 | 370 | 408 | 247 | 326 | 330 | 386 | 344 | 401 |
| 300 | 445 | 494 | 297 | 395 | 408 | 466 | 418 | 484 |
| 350 | 516 | 574 | 345 | 459 | 444 | 539 | 486 | 560 |
| 400 | 590 | 661 | 394 | 529 | 527 | 623 | 561 | 650 |
| 450 | 668 | 744 | 446 | 595 | 596 | 688 | 633 | 735 |
| 500 | 743 | 826 | 496 | 660 | 675 | 775 | 697 | 810 |
| 1000 | 1494 | 1658 | 996 | 1326 | 1327 | 1546 | 1408 | 1629 |
| 1500 | 2243 | 2491 | 1496 | 1992 | 1975 | 2327 | 2110 | 2457 |
| 2000 | 2992 | 3326 | 1996 | 2660 | 2672 | 3106 | 2834 | 3277 |
| 2500 | 3740 | 4159 | 2494 | 3327 | 3338 | 3876 | 3527 | 4090 |
| 3000 | 4494 | 4989 | 2996 | 3991 | 3997 | 4663 | 4235 | 4902 |
| 3500 | 5238 | 5823 | 3493 | 4658 | 4711 | 5425 | 4921 | 5729 |
| 4000 | 5989 | 6658 | 3993 | 5326 | 5330 | 6196 | 5648 | 6538 |
| 4500 | 6743 | 7491 | 4496 | 5993 | 6000 | 7005 | 6377 | 7362 |
| 5000 | 7489 | 8325 | 4994 | 6660 | 6665 | 7774 | 7038 | 8185 |

Table 3.1: Insert experiment key comparison costs for Figures 3.1–3.8

## *3.3 Modified Insert Experiments*

This section details the number of node-to-node key comparisons required by both the 2-3 heap and 3-4 heap during insert operations. Unlike the previous insert experiments, both data structures have been modified with an insertion cache.

This involved creating a non-adaptive cache on the incoming stream of nodes in an effort to reduce the number of node-to-node key comparisons required to insert the new node into its correct tree position. The design involves caching up to four consecutive nodes, three for 2-3 heap, and each new node will be compared with the largest node held by the cache. When the cache reaches capacity it will flush into the T(1) top level position tree, but if the new node is smaller than the largest held by the cache, then the cache will flush into the $T(0)$ top level position tree and the newest node shall remain behind in the cache. Under ideal conditions the caching mechanism will give a constant node-to-node comparison cost of zero point seven five (0.75) to insert the new node into their correct position on the $T(0)$ tree. Under non-ideal conditions, the caching mechanism will be a liability by adding one additional node-to-node comparison cost on top of standard node insertion.

Different insertion experiments were performed where the values inserted were a) incremented by one (1), b) decremented by one (1), c) random number generated using modulus of five hundred (500), and d) using a series of four numerical values where the series seed is randomly generated using modulus of five hundred (500) followed by three increments of one (1).

Experiment results data used to generate each figure presented in this section are shown in Table 3.2 on page 55.

### *3.3.1 Increasing Numerical*

Inserting increasing numerical data will generate a small number of key comparisons because each node key value is larger than that previously inserted. Each new node will therefore get inserted as the insertion caches largest node. Since the cache will never flush prematurely it will always reach full capacity and these flushed nodes will be inserted into the T(1) top level position tree. Once the cache has been flushed the next node to be inserted will incur no

key comparison cost. For every four nodes inserted, the insertion cache will perform only three key comparisons, which is a reduction in key comparisons used by forty (40) percent. Likewise for the 2-3 heap with three nodes inserted, the insertion cache will perform only two key comparisons, which is a reduction in key comparisons used by thirty three (33) percent.

Because both heaps have very similar operating characteristics, their respective key comparison cost increased at a constant rate. Inspecting the graphs it becomes obvious that both heaps had virtually identical performance. Of the eighteen experiments performed, 3-4 heap out performed 2-3 heap eight times, performed equally three times, and performed worse on seven occasions.

Inspecting the insertion of twelve (12) nodes only, without the insertion cache the 2-3 heap and 3-4 heap would have respectively used twelve (12) and fifteen (15) key comparisons at the $T(0)$ level. With the insertion cache, key comparisons used reduced to eight (8) and nine (9) respectively. The closeness of this experiment results came as an unexpected surprise because it was expected both heaps would perform equally well and therefore the 3-4 heap would have consistently a higher key comparison cost.

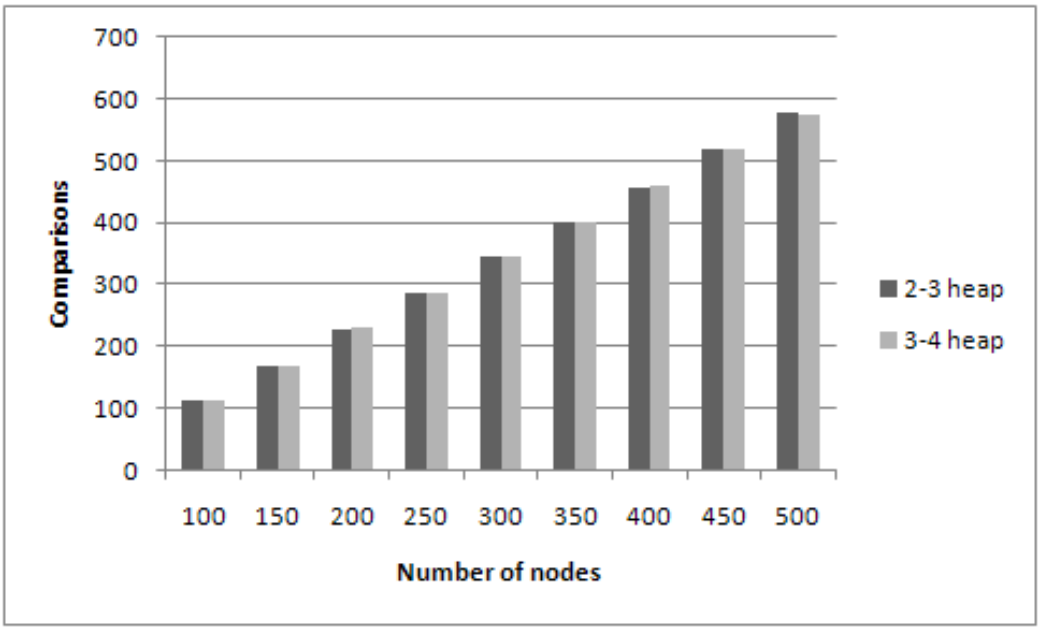


Figure 3.9: Insert increasing numerical value using modified 2-3 heap and 3-4 heap

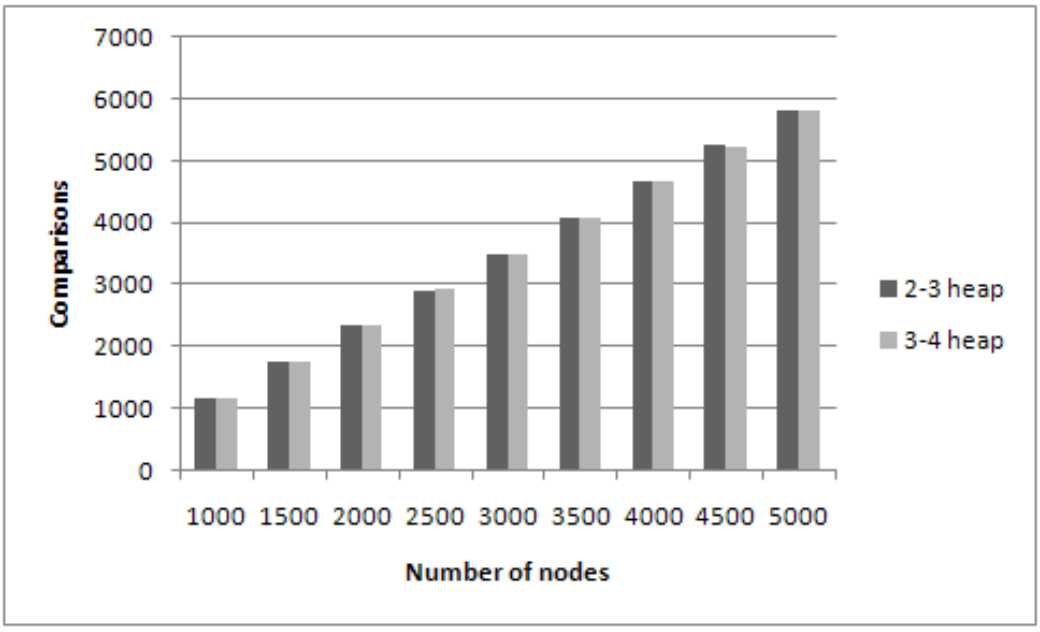


Figure 3.10: Insert increasing numerical value using modified 2-3 heap and 3-4 heap

### *3.3.2 Decreasing Numerical*

Inserting decreasing numerical data will generate a large number of key comparisons because the number of nodes in the insertion cache will never exceed one and will constantly incur an additional single key comparison cost per node inserted. The previously insert node would then get flushed out of the cache and be inserted at the front of a $T(0)$ top level position tree. Both heaps have very similar operating characteristics and their respective key comparison cost increase at a constant rate. However the 3-4 heap required more key comparisons than the 2-3 heap. As noted in Section 3.2.1, the 3-4 heap uses a flat cost of two comparisons to determine where on the trunk the fourth node insertion point and because of this fixed comparison cost, this makes the insertion of the fourth node more expensive than the most efficient possible cost achieved by linear comparison scanning.

Comparing the insertion of decreasing numerical values with / without an insertion cache, it can be seen that with the insertion cache the total key comparison cost is larger by approximately the number of nodes being inserted. This essentially means the insertion cache did not aid in reducing the number of key comparisons because it was given its worst case type of insertion numerical values.

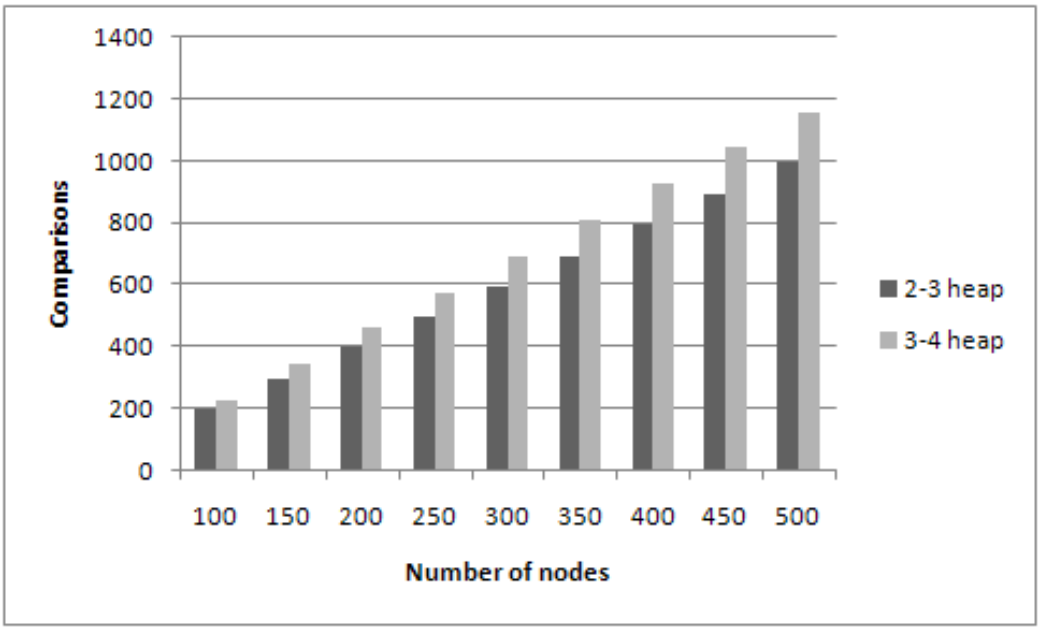


Figure 3.11: Insert decreasing numerical value using modified 2-3 heap and 3-4 heap

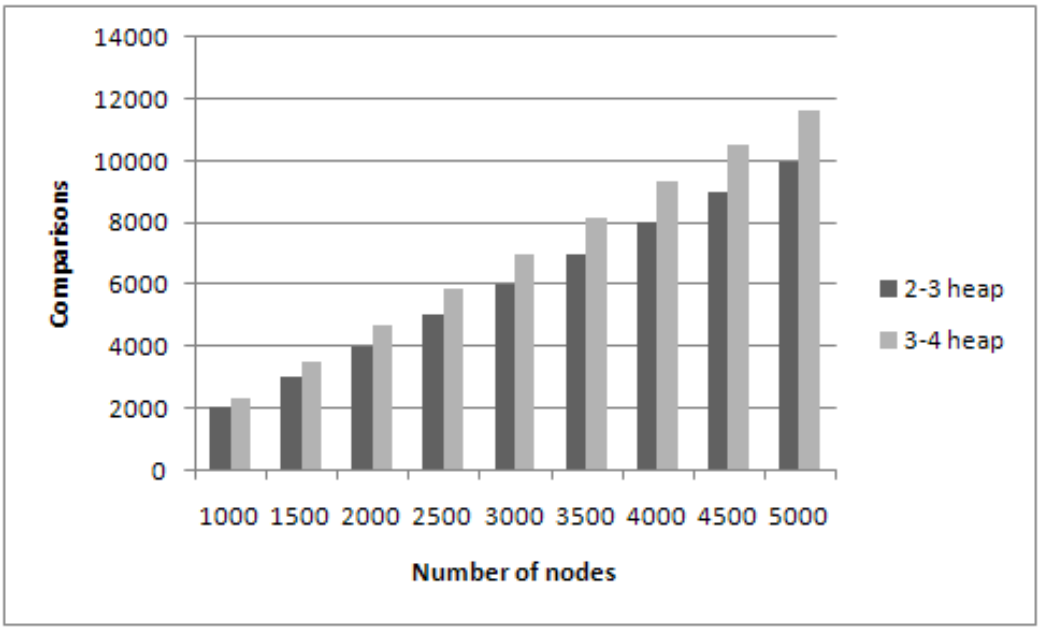


Figure 3.12: Insert decreasing numerical value using modified 2-3 heap and 3-4 heap

### 3.3.3 *Random Numerical*

Inserting random numerical data will generate key comparisons based upon the insertion of numerical values occurring where the next value is randomly generated without any series pattern and this could cause the cache to flush itself frequently into the $T(0)$ top level position tree. Because both heaps have very similar operating characteristics, their respective key comparison cost increased at a constant rate. However the 3-4 heap required more key comparisons than the 2-3 heap as noted in Section 3.2.1.

When comparing with the results of inserting random numerical data without an insertion cache, Section 3.2.3, the key comparison cost was smaller without the insertion cache and therefore performance was better. This deterioration in performance was due to the insertion cache needing to flush itself when the newest node key value was smaller than the largest node key value held inside the insertion cache. The act of flushing the cache would have incurred a single key comparison penalty in comparison to when the cache had simply reached full capacity.

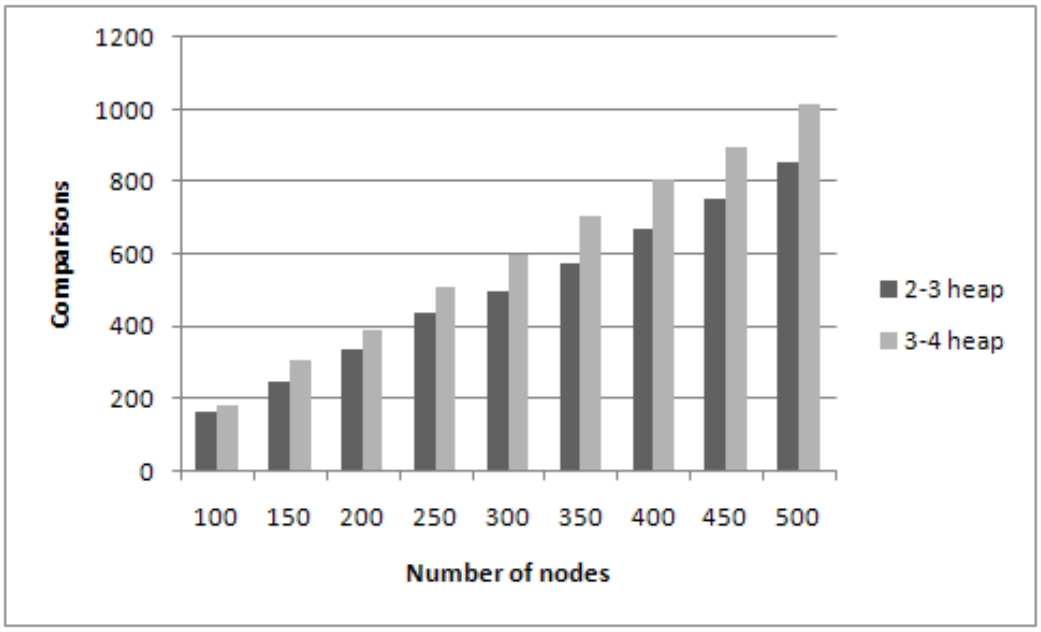


Figure 3.13: Insert randomly generated modulus 500 numerical value using modified 2-3 heap and 3-4 heap

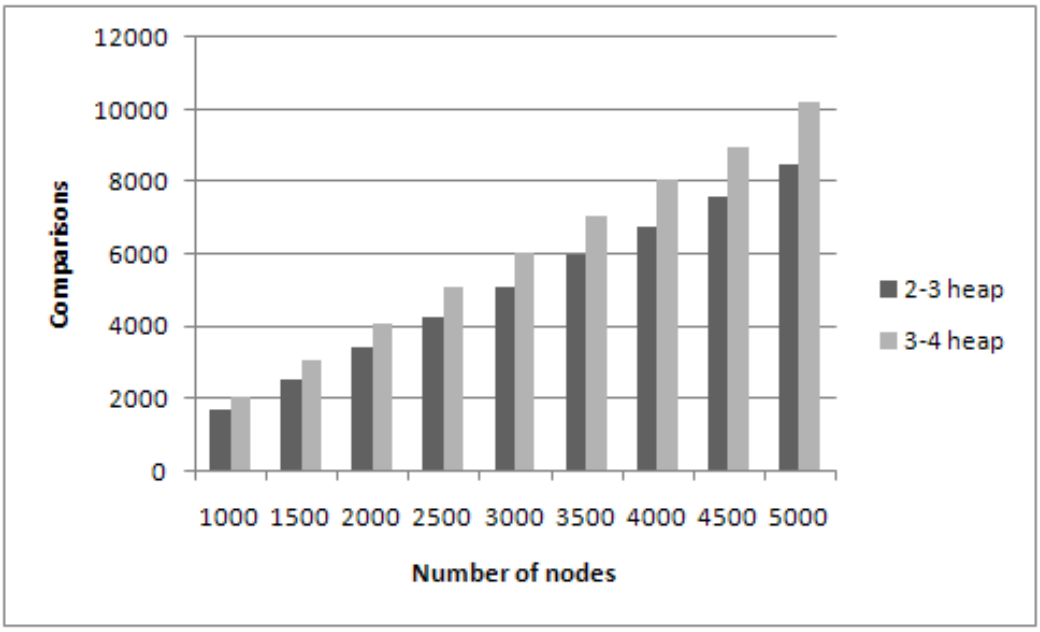


Figure 3.14: Insert randomly generated modulus 500 numerical value using modified 2-3 heap and 3-4 heap

### *3.3.4 Osculating Numerical*

Inserting osculating numerical data will generate key comparisons based upon the insertion of a series being incremented by one (1) consecutively three times before a new series seed was randomly generated with modulus five hundred (500).

Inspecting Figures 3.15–3.16, it can be seen that the 3-4 heap outperformed the 2-3 heap for every single experiment size by approximately 9 percent. The reason this phenomenon occurs lies in the relationship between the 3-4 heap insertion cache and the osculating nature of the insertion data.

The 3-4 heap insertion cache is capable of holding four nodes before they are flushed into the $T(1)$ top level position tree because the numerical data being inserted was incremented consecutively three times after the series seed value was generated. This osculating pattern incidentally allowed the 3-4 heap insertion cache to perform at its optimal whilst the 2-3 heap had to prematurely flush its cache periodically and incur additional key comparison cost penalties. This comparison cost penalty is incurred because the next

insert stream node key value was not larger than the largest node key value held by the cache.

For the 3-4 heap, if the seed value was twenty five then the 3-4 heap insertion cache would have held at time of flushing nodes with key values of twenty five, twenty six, twenty seven and twenty eight. The cache was full when flushed and because the cache was empty when the new series generated node was inserted, no key comparisons were performed. This meant that for every four nodes inserted, the insertion cache performed only three key comparisons.

For the 2-3 heap under the same scenario, it would have flushed the cache when it held three nodes with key values of twenty five, twenty six and twenty seven. The insertion cache would have used two key comparisons. The next node to be inserted into the cache has a key value of twenty eight, thus concluding this series. If the new series seed value was larger than twenty eight, then the insertion cache would not have flushed itself but if the new series seed value was smaller than twenty eight, then the cache would have to flush itself and subsequently incur a single key comparison penalty in relation to when the cache had simply reached full capacity.

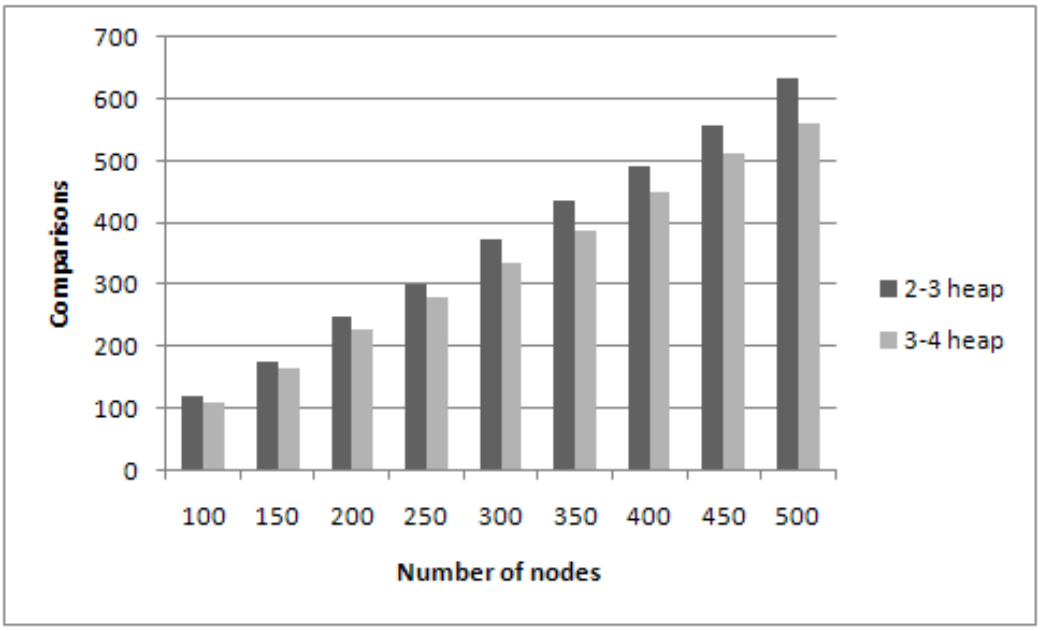


Figure 3.15: Insert osculating generated modulus 500 numerical value using modified 2-3 heap and 3-4 heap

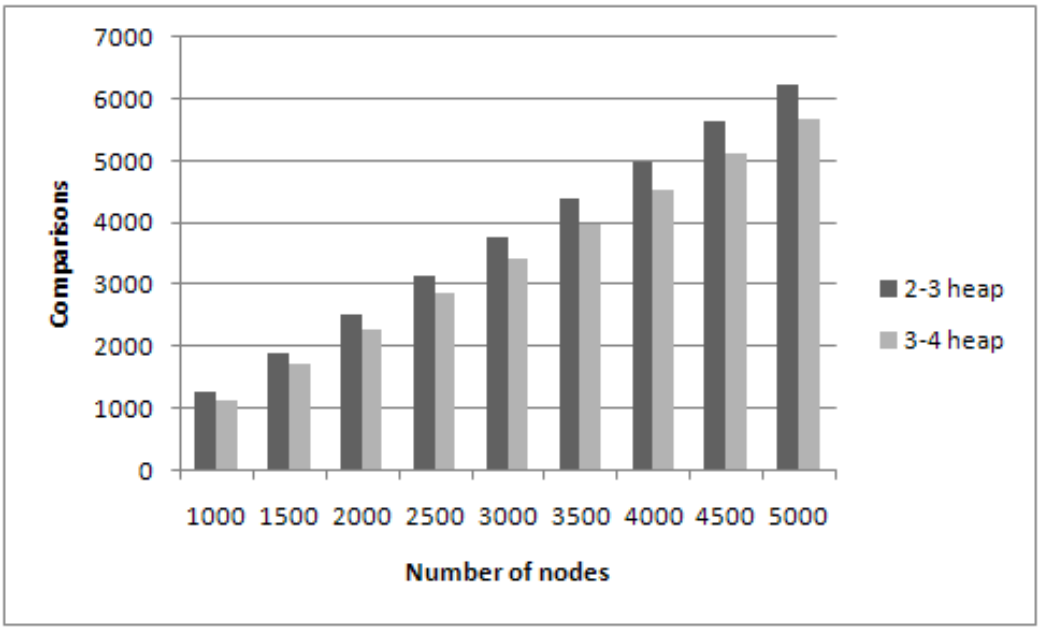


Figure 3.16: Insert osculating generated modulus 500 numerical value using modified 2-3 heap and 3-4 heap

| Number | Increasing | | Decreasing | | Random | | Osculating | |
|---|---|---|---|---|---|---|---|---|
| of Nodes | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap |
| 100 | 112 | 111 | 196 | 226 | 161 | 181 | 119 | 109 |
| 150 | 168 | 168 | 293 | 342 | 243 | 303 | 173 | 164 |
| 200 | 227 | 229 | 394 | 459 | 336 | 391 | 247 | 225 |
| 250 | 287 | 284 | 496 | 574 | 434 | 507 | 301 | 277 |
| 300 | 345 | 344 | 595 | 691 | 494 | 596 | 373 | 334 |
| 350 | 400 | 400 | 693 | 807 | 576 | 707 | 437 | 386 |
| 400 | 457 | 461 | 793 | 924 | 668 | 807 | 490 | 450 |
| 450 | 518 | 520 | 892 | 1043 | 754 | 895 | 557 | 511 |
| 500 | 577 | 576 | 994 | 1157 | 854 | 1017 | 635 | 560 |
| 1000 | 1161 | 1158 | 1995 | 2322 | 1681 | 2012 | 1247 | 1129 |
| 1500 | 1743 | 1741 | 2993 | 3488 | 2533 | 3057 | 1874 | 1707 |
| 2000 | 2326 | 2326 | 3994 | 4655 | 3403 | 4044 | 2493 | 2277 |
| 2500 | 2907 | 2909 | 4993 | 5824 | 4253 | 5054 | 3123 | 2840 |
| 3000 | 3494 | 3489 | 5994 | 6987 | 5074 | 6030 | 3745 | 3402 |
| 3500 | 4072 | 4073 | 6991 | 8154 | 5941 | 7069 | 4387 | 3979 |
| 4000 | 4656 | 4658 | 7992 | 9320 | 6769 | 8086 | 4980 | 4538 |
| 4500 | 5243 | 5241 | 8992 | 10490 | 7553 | 8966 | 5642 | 5112 |
| 5000 | 5823 | 5825 | 9992 | 11656 | 8486 | 10202 | 6244 | 5685 |

Table 3.2: Insert experiment key comparison costs for Figures 3.9–3.16

### *3.3.5 Conclusion*

For experiments run without the insertion cache, it can be seen that the 3-4 heap key comparison cost is always higher than the 2-3 heap. Whilst being higher, the results generated by both heaps display a linear complexity of $O(m)$. The key comparison cost difference was most profound whilst inserting continuously decreasing data. For the other three types of insertion data, the key comparison costs incurred were of similar magnitudes.

For experiments run with the insertion cache, it can be seen that the 3-4 heap key comparison cost would equal and even be less than that generated by the 2-3 heap. This can be seen with the insertion of increasing and osculating data. If the insertion of random data is assumed to replicate actual usage conditions, then the insertion cache does not provide an overall reduction in key comparisons used.

## *3.4 Delete-min Experiments*

This section details the number of node-to-node key comparisons required by both the 2-3 heap and 3-4 heap during delete-min operations. Once the dataset has been inserted into the heaps, their respective key comparison cost totals were reset to zero so the delete-min results were not slightly askew because of differing insertion key comparison costs. The results of these experiments are expected to display the $O(n \log\ n)$ profile.

Different delete-min experiments were performed where the dataset was configured by inserting values which were a) incremented by one (1), b) decremented by one (1), and c) randomly generated using modulus of five hundred (500).

Experiment results data used to generate each figure presented in this section are shown in Table 3.3 on page 63.

### *3.4.1 Increasing Numerical*

Delete-min with increasing numerical points inserted into the dataset where each new point is incremented by one over the previously inserted point. The results charted by Figures 3.17–3.18 are inconsistent and do not portray any pattern, particularly not the expected $O(n \log\ n)$ profile.

Of the twenty experiments performed, the 3-4 heap had the lowest key comparison cost in twelve, and nearly equalled in another. Reflecting upon the insertion experiments performed in Section 3.2, the standard 3-4 heap consistently incurred the highest key comparison cost so this heap feature cannot be used to explain why it had the lowest key comparison cost in most experiments. Both heaps are also standard so there are no modifications to consider. There is only one other fundamental difference between both heaps which could have an impact. For each 3-4 heap tree in top level position $T(i)$, it can contain more nodes that the 2-3 heap counterpart. This results in fewer trees and with less top level positions occupied, there would be less key comparisons being performed to locate the smallest key valued node for deletion.

To confirm this, these experiments were performed again but excluded all key comparisons generated from the handling of sub-trees $\mathbf{b_j}$ ($j\ =\ 0,\ \ldots,\ i-$

1) (post the removal of the smallest key valued node). These results can be seen in Figures 3.19–3.20 and display the expected $O(n \log\ n)$ profile curve with the 3-4 heap consistently having the lowest key comparison cost.

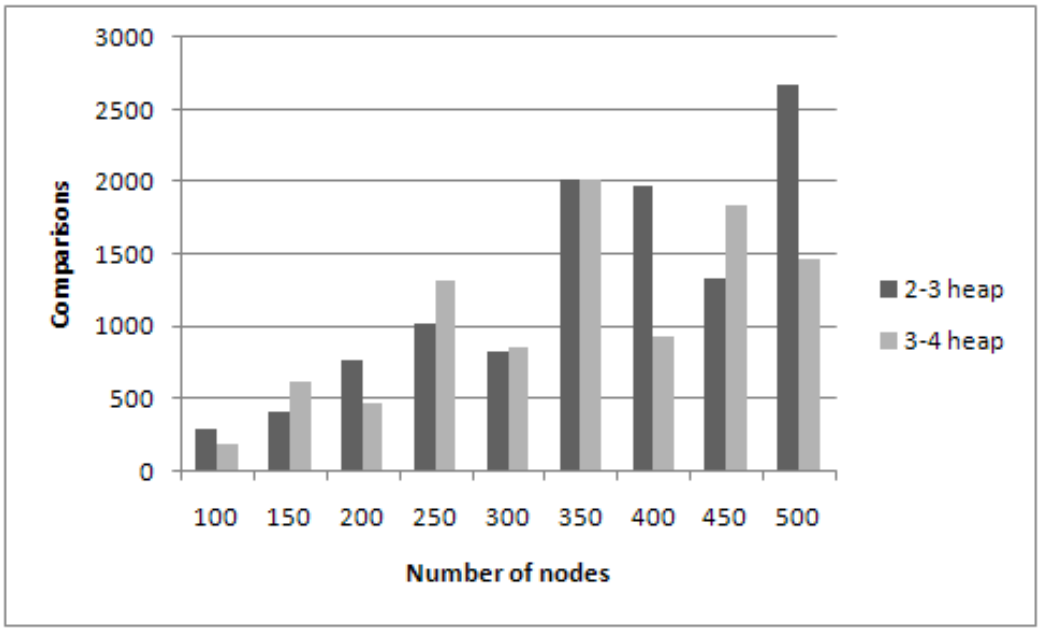


Figure 3.17: Delete-min where nodes were inserted with increasing numerical value

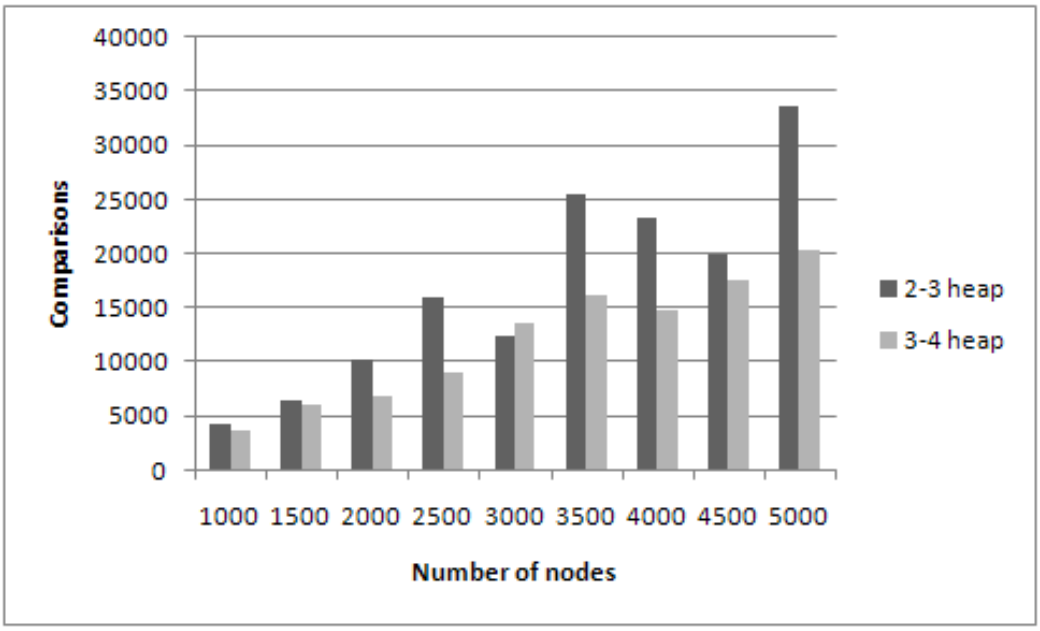


Figure 3.18: Delete-min where nodes were inserted with increasing numerical value

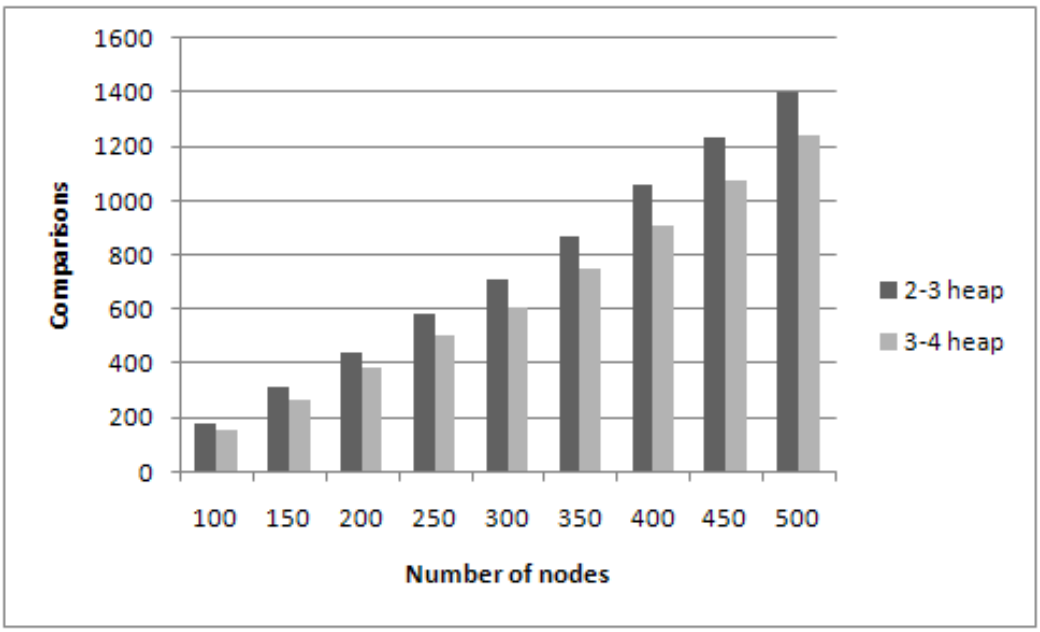


Figure 3.19: Delete-min where nodes were inserted with increasing numerical value, excluding all key comparisons generated from the handling of sub-trees $\mathbf{b_j}$ $(j = 0, \ldots, i-1)$

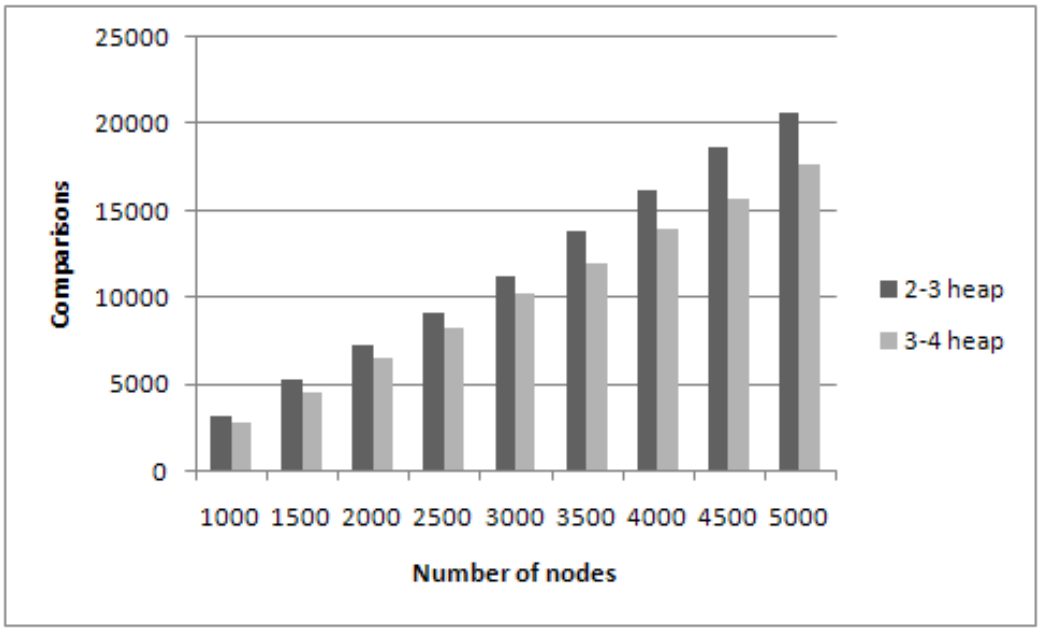


Figure 3.20: Delete-min where nodes were inserted with increasing numerical value, excluding all key comparisons generated from the handling of sub-trees $\mathbf{b_j}$ $(j = 0, \ldots, i-1)$

### *3.4.2 Decreasing Numerical*

Delete-min with decreasing numerical points inserted into the dataset where each new point is decremented by one over the previously inserted point. Both heaps have a very similar operating characteristic and their respective key comparison cost increased at a constant $O(n \log\ n)$ rate, however the 3-4 heap consistently incurred less key comparisons as seen in Figures 3.21–3.22. These results include key comparisons incurred from handling sub-trees $\mathbf{b_j}$ $(j = 0, \ldots, i-1)$.

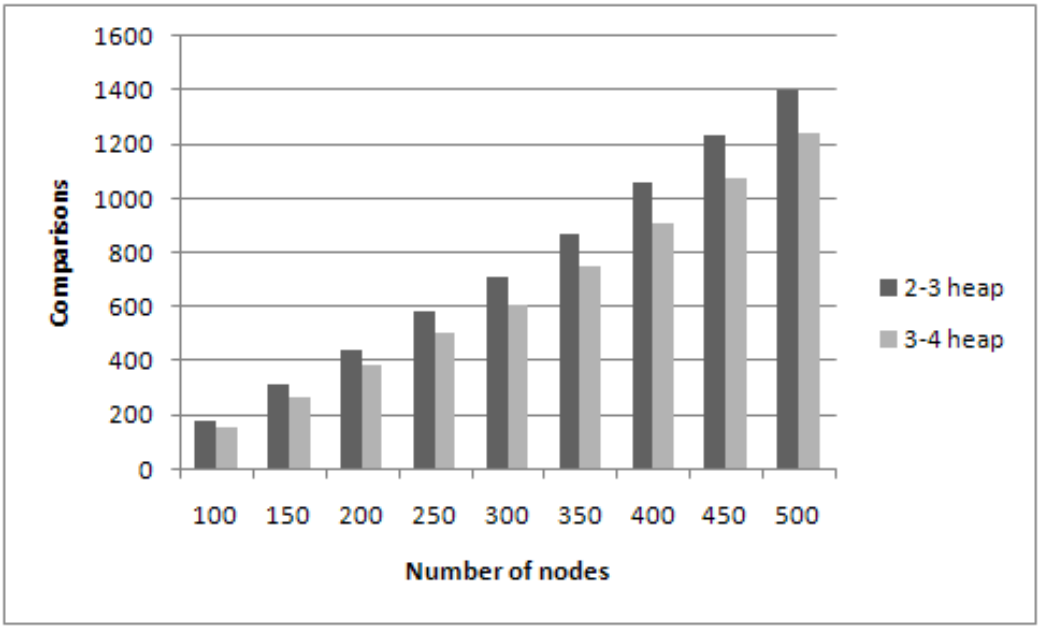


Figure 3.21: Delete-min where nodes were inserted with decreasing numerical value

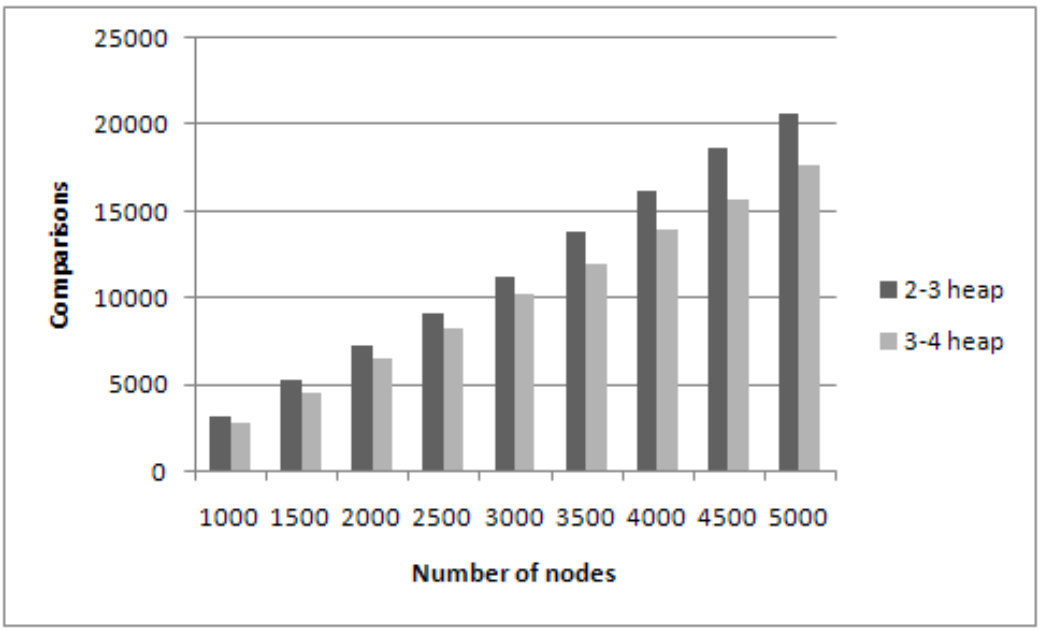


Figure 3.22: Delete-min where nodes were inserted with decreasing numerical value

### *3.4.3 Random Numerical*

Delete-min with random numerical points inserted into the dataset where each new point is randomly generated using modules five hundred (500). Both heaps have a very similar operating characteristic and their respective key comparison cost increased at a constant $O(n \log\ n)$ rate, however the 3-4 heap consistently incurred less key comparisons as seen in Figures 3.23–3.24. These results include key comparisons incurred from handling sub-trees $\mathbf{b_j}$ $(j\ =\ 0,\ \ldots,\ i-1)$.

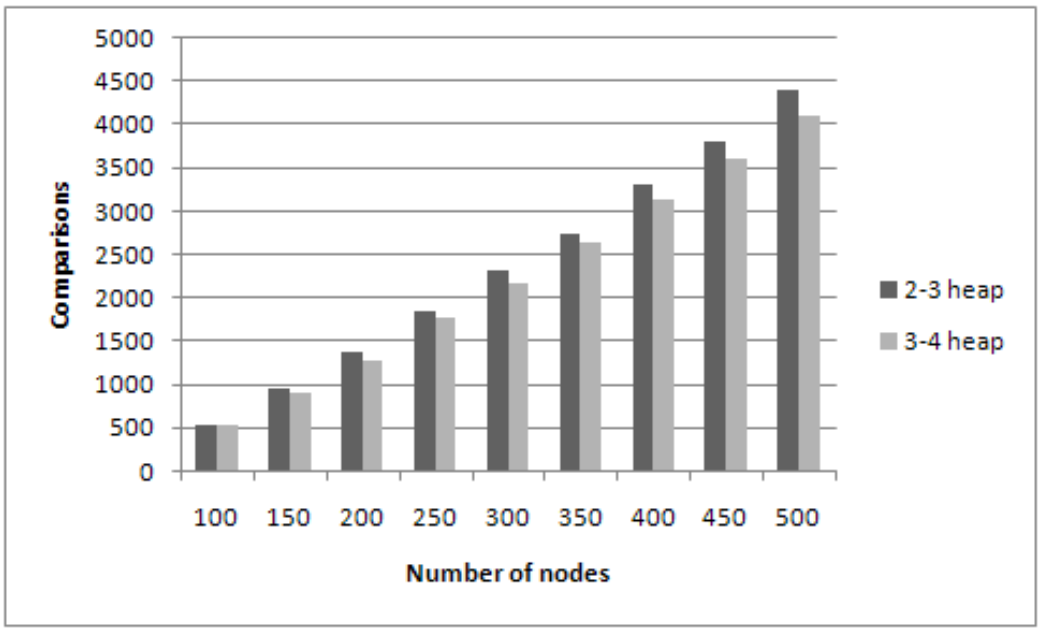


Figure 3.23: Delete-min where nodes were inserted with random numerical value

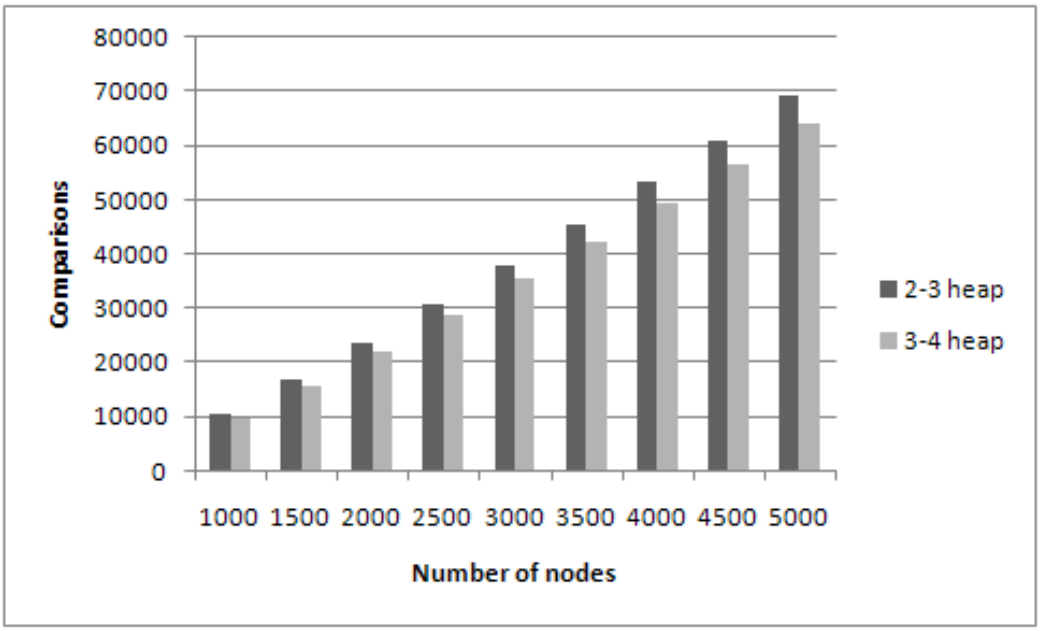


Figure 3.24: Delete-min where nodes were inserted with random numerical value

### *3.4.4 Conclusion*

The 3-4 heap consistently incurred the least number of key comparisons because for each 3-4 heap tree in top level position $T(i)$, it can contain more nodes that the 2-3 heap counterpart. This results in fewer trees and with less top level positions occupied there would be less key comparisons being performed to locate the smallest key valued node for deletion. Both heaps exhibited a key comparison cost which increased at a constant complexity of $O(n \log\ n)$ rate.

The number of key comparisons for insertions in increasing order is greater than decreasing order or random order. This is because in increasing order, the smallest key valued node can be found more likely in larger trees and the merging of sub-trees $\mathbf{b_j}$ ($j\ =\ 0,\ \ldots,\ i-1$) will incur more key comparisons. This is because there are more sub-trees to handle and the likelihood of the lower top level positions already containing a tree is higher.

| Number | Increasing | | Inc. $O(n \log\ n)$ | | Decreasing | | Random | |
|---|---|---|---|---|---|---|---|---|
| of Nodes | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap |
| 100 | 286 | 178 | 172 | 154 | 172 | 154 | 521 | 512 |
| 150 | 408 | 613 | 314 | 264 | 314 | 264 | 934 | 893 |
| 200 | 756 | 456 | 435 | 380 | 435 | 380 | 1356 | 1276 |
| 250 | 1010 | 1312 | 578 | 499 | 578 | 499 | 1847 | 1754 |
| 300 | 824 | 845 | 710 | 608 | 710 | 608 | 2315 | 2162 |
| 350 | 2011 | 2022 | 865 | 749 | 865 | 749 | 2747 | 2634 |
| 400 | 1963 | 931 | 1062 | 907 | 1062 | 907 | 3310 | 3135 |
| 450 | 1328 | 1831 | 1234 | 1078 | 1234 | 1078 | 3806 | 3601 |
| 500 | 2668 | 1469 | 1403 | 1239 | 1403 | 1239 | 4393 | 4094 |
| 1000 | 4141 | 3547 | 3084 | 2734 | 3084 | 2734 | 10209 | 9609 |
| 1500 | 6466 | 5882 | 5201 | 4450 | 5201 | 4450 | 16683 | 15610 |
| 2000 | 10088 | 6674 | 7164 | 6444 | 7164 | 6444 | 23599 | 21959 |
| 2500 | 15923 | 8881 | 9083 | 8191 | 9083 | 8191 | 30624 | 28538 |
| 3000 | 12301 | 13594 | 11244 | 10191 | 11244 | 10191 | 37823 | 35271 |
| 3500 | 25532 | 16215 | 13750 | 11930 | 13750 | 11930 | 45549 | 42194 |
| 4000 | 23214 | 14737 | 16172 | 13924 | 16172 | 13924 | 53341 | 49387 |
| 4500 | 19860 | 17574 | 18595 | 15660 | 18595 | 15660 | 60977 | 56581 |
| 5000 | 33590 | 20310 | 20654 | 17662 | 20654 | 17662 | 69138 | 64027 |

Table 3.3: Delete-min experiment key comparison costs for Figures 3.17–3.24

### *3.5 Decrease-key Experiments*

This section details the number of node-to-node key comparisons required by both the 2-3 heap and 3-4 heap during decrease-key operations.

The setup of the following decrease-key experiments had each dataset point value incremented by two (2) over the previously inserted point value, the initial point value was twenty (20). Once the dataset has been inserted into the heaps, their respective key comparison cost totals were reset to zero so the decrease-key results will not be slightly askew because of differing insertion key comparison costs. Using this configuration, four (4) different experiments were performed whereby a node's key value was reduced by one (1) a) sequentially in order of insertion from '$n$' to first, b) sequentially in order of insertion from first to '$n$', and c) randomly select nodes until '$n$' decrease-key operations have been performed. For experiment d) a node's key value was reduced sequentially in order of insertion from '$n$' to first except that it is executed three times with the node's key value decreased respectively by one (1), two (2) or three (3). This ensures each node was decremented by a value which was 'less than', 'equal to', or 'greater than'

the gap of two (2) with its immediate neighbour above.

For experiment suite d), both heaps had identical modifications made to their decrease-key process. These modifications were such that when a node's key value was decreased, it was not automatically removed from its current position unless its value was smaller than the immediate neighbour above. The aim of this modification is to reduce the number of node-to-node key comparisons performed by bypassing the standard decrease-key node removal and insertion process. One disadvantage of this approach is that if the node does require removal from its current location, an additional node-to-node key comparison cost has been incurred in addition to those incurred by the standard decrease-key node removal and insertion process.

Figures 3.31–3.34 chart the modified and standard results side-by-side respectively for 2-3 heap and 3-4 heap. Experiment results data used to generate each figure presented in this section are shown in Tables 3.4–3.6 on pages 73–75.

### *3.5.1 Sequentially from 'n' to first*

Decrease-key sequentially from '$n$' to first node inserted into the heap incurred the smallest key comparison cost for the 2-3 heap whilst nearly the highest key comparison cost for 3-4 heap. Both heaps have very similar operating characteristics and it observed that their respective key comparison cost increased at a constant linear complexity $O(m)$, however the 3-4 heap required $2\frac{1}{2}$ times more key comparisons than 2-3 heap because the 2-3 heap key comparison cost averages to less than one versus just over two key comparison per node.

The semi-rigid nature of the heaps trunk where the trunk size can grow and shrink by one, requires a higher key comparison cost on the 3-4 heap than the 2-3 heap because when a trunk was one node to short, the 2-3 heap would require one (1) key comparison versus two (2) for the 3-4 heap in order to restore standard arrangement for that particular trunk. This leaves the 3-4 heap at a disadvantage with a higher key comparison cost during all decrease-key operations which require restructuring operations to restore standard arrangement.

There are two scenarios where a heap would not incur any key comparisons. The first scenario is when the node being removed leaves the tree within the boundaries of standard arrangement, or no key comparisons were incurred during rearrangement back into standard arrangement. With the node now removed from its $T(i)$ tree it must be inserted into the appropriate $T(0), \ \ldots, \ T(i-1)$ tree, and that top level position must be presently empty. The second scenario is when the node having the decrease-key performed upon itself, is located as the root node of a tree. Under this situation there is no requirement to remove the node because its already got the smallest key node value for the tree in top level position $T(i)$.

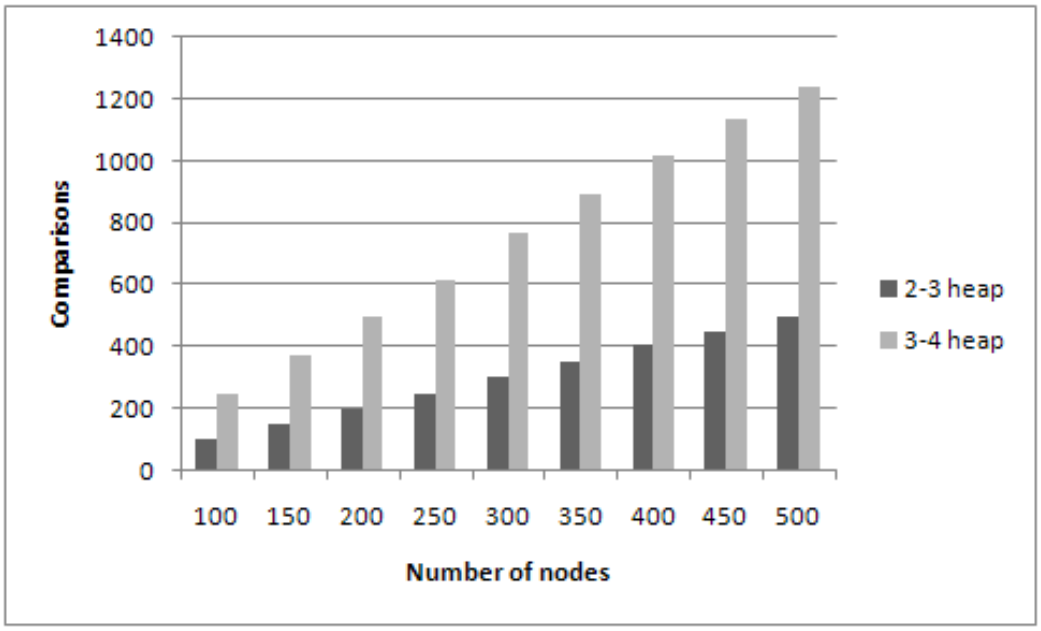


Figure 3.25: Decrease-key sequentially from '$n$' to first

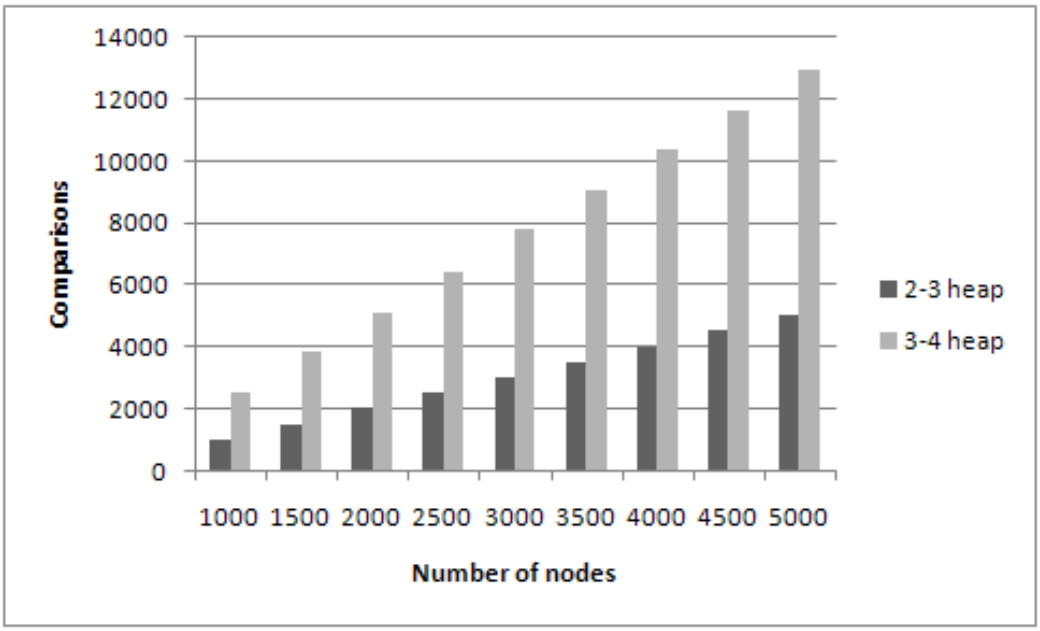


Figure 3.26: Decrease-key sequentially from '$n$' to first

### *3.5.2 Sequentially from first to 'n'*

Decrease-key sequentially from first to $n$-th node inserted into the heap incurred the largest key comparison cost for both the 2-3 heap and 3-4 heap. Both heaps have very similar operating characteristics and it observed that their respective key comparison cost increased at a constant linear complexity $O(m)$; however the 3-4 heap required more key comparisons than 2-3 heap.

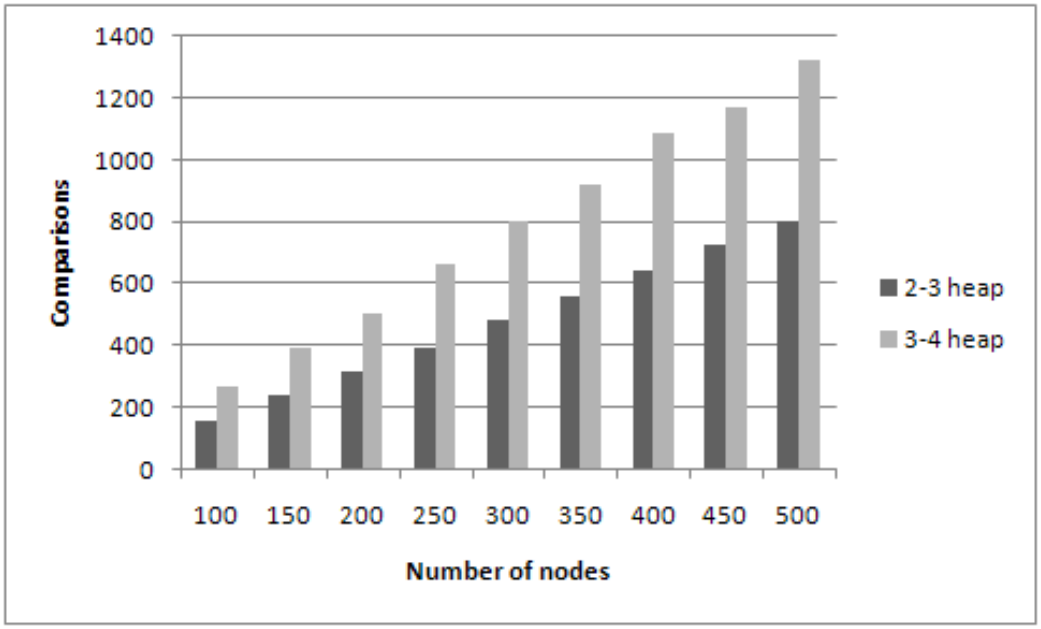


Figure 3.27: Decrease-key sequentially from first to '$n$'

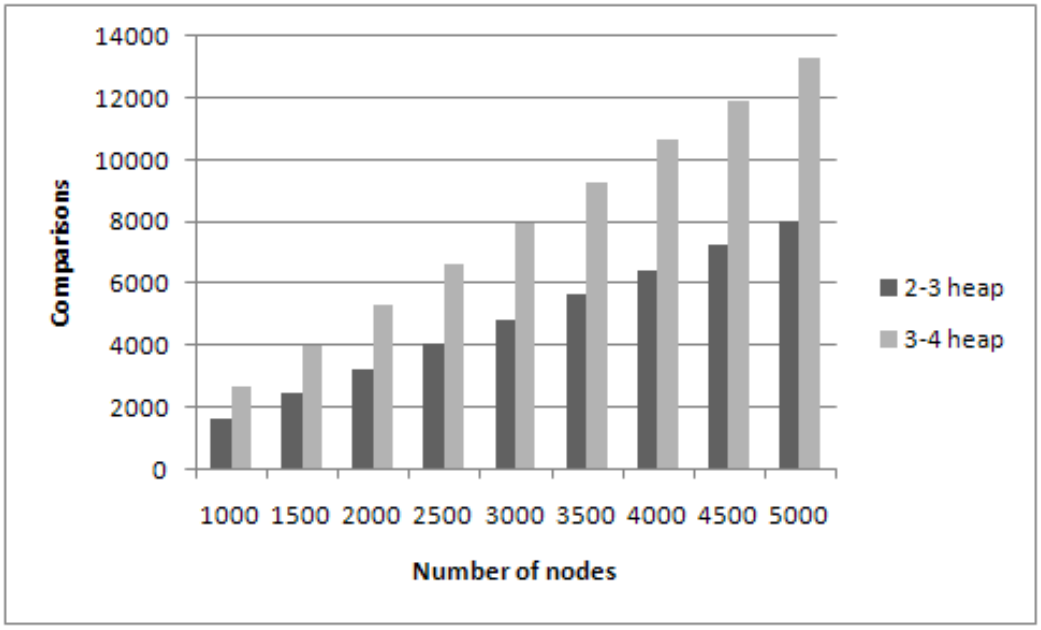


Figure 3.28: Decrease-key sequentially from first to '$n$'

### *3.5.3 Randomly*

Decrease-key randomly for '$n$' times on randomly selected nodes which have been inserted into the heap, this experiment incurred approximately the middle key comparison cost for the 2-3 heap whilst had the lowest key comparison cost for 3-4 heap. Both heaps have very similar operating characteristics and it observed that their respective key comparison cost increased at a constant linear complexity $O(m)$; however the 3-4 heap required more key comparisons than 2-3 heap.

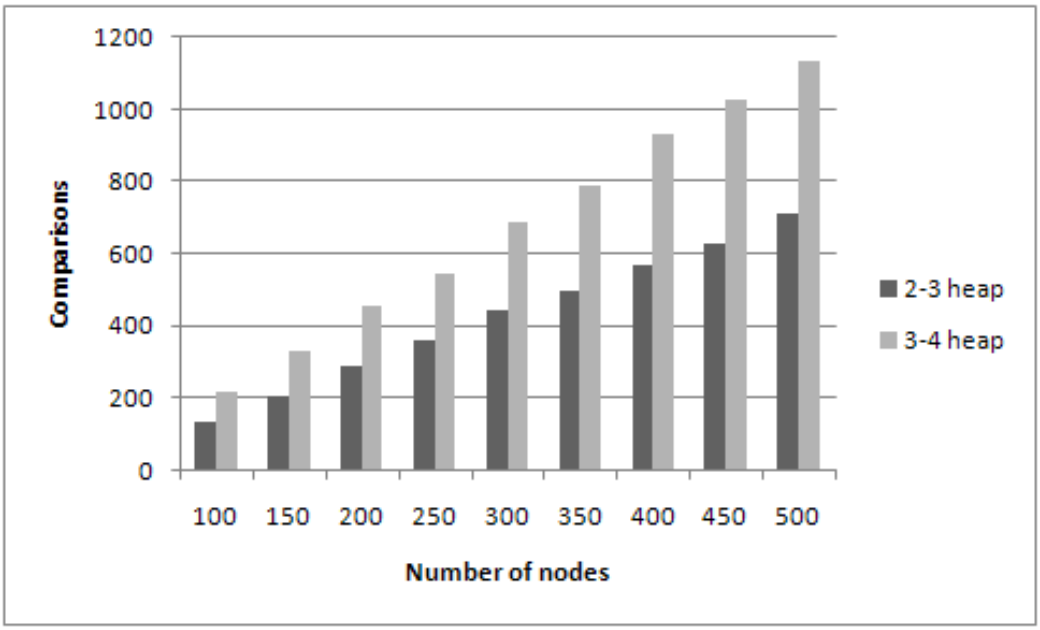


Figure 3.29: Decrease-key randomly

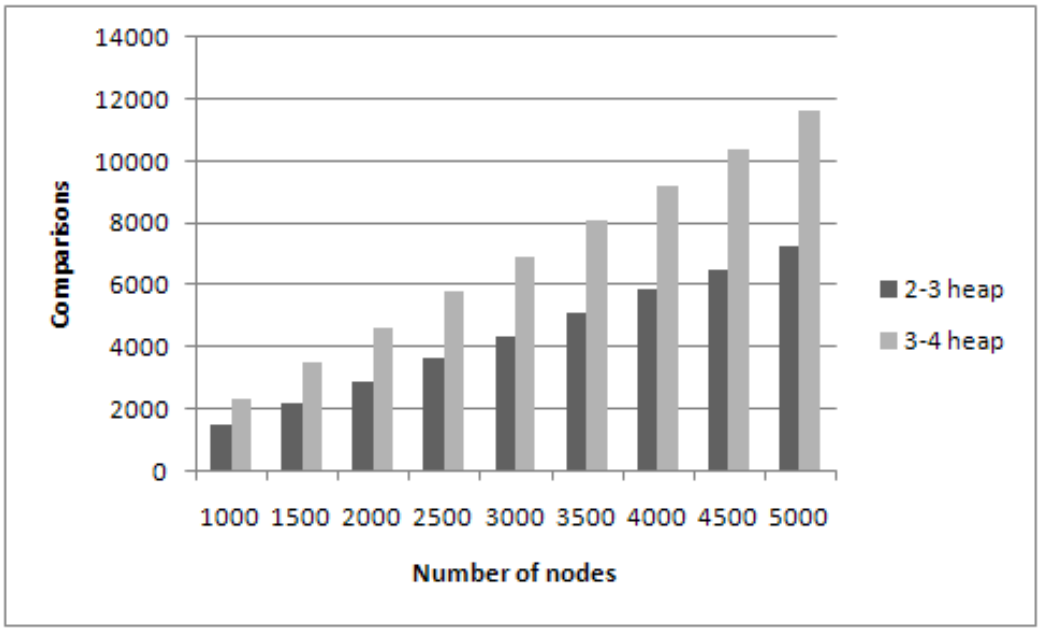


Figure 3.30: Decrease-key randomly

### *3.5.4 Methodically*

Decrease-key methodically generates key comparison data for node key value reduction where the 2-3 heap and 3-4 heap respectively have their decrease-key process standard and modified. The heaps underwent identical modifications such that when a node's key value is decreased, it will not be automatically removed from its current position unless its value was smaller than the node immediate neighbour above. The aim and advantage of this modification is to reduce the number of node-to-node key comparisons performed by bypassing the standard decrease-key node removal and insertion process. One disadvantage when a node does require usage of the standard decrease-key process, there has been incurred one additional key comparison. In the following figures, the horizontal axis legend labels for the modified heap results have been identified by post-pending '*Mod*'.

For these experiments, a node's key value was reduced sequentially in order of insertion from '$n$' to first except that each experiment was performed three times with the node's key value decreased respectively by one (1), two (2) and three (3). This ensures each node was decremented by a value

which was 'less than', 'equal to', or 'greater than' the gap of two (2) with its immediate neighbour above.

The predicted results from this experiment for the standard heaps are that their key comparison costs will be essentially identical for all decrease-key values. For the modified heaps, their key comparison costs will be markedly reduced for decrease-key values one (1) and two (2) since the node does not get removed from its current location, whilst increase markedly for decrease-key value three (3) because the node under goes the standard decrease-key process and also incurred an additional key comparison cost from the modification. However the actual results did not completely reflect this predicted outcome.

Inspecting the 2-3 heap results, for decrease-key values one (1) and two (2) the key comparison cost was unexpectedly essentially unchanged when compared against results without the decrease-key modification. This means the key comparison cost generated by the modified process was identical to the standard process. For decrease-key of value three (3), the key comparison cost was as expected higher than the results without the decrease-key modification. This means the key comparison cost of the modification was added on top of the standard process key comparison cost.

Inspecting the 3-4 heap results, for decrease-key values one (1) and two (2) the key comparison cost was reduced by half when compared against results without the decrease-key modification and was essentially identical to results generated by the 2-3 heap. This means the key comparison cost generated by the modified process was half that of the standard process. For decrease-key of value three (3), the key comparison cost was unexpectedly slightly less than results from without the decrease-key modification. Investigating into this further, it was identified that during a decrease-key operation the tree would occasionally require rearrangement to get itself back into standard arrangement. During this process the gap between nodes can increase to become more than two (2) and if the node in this situation had a decrease-key operation performed against it, it shall remain in position because its key value was smaller than the immediate neighbour above. This means that only a single key comparison cost has been incurred which is less than would have been during the standard process. If this scenario occurs frequently enough,

the net amount of key comparisons incurred with the modified process would be less than without this modification.

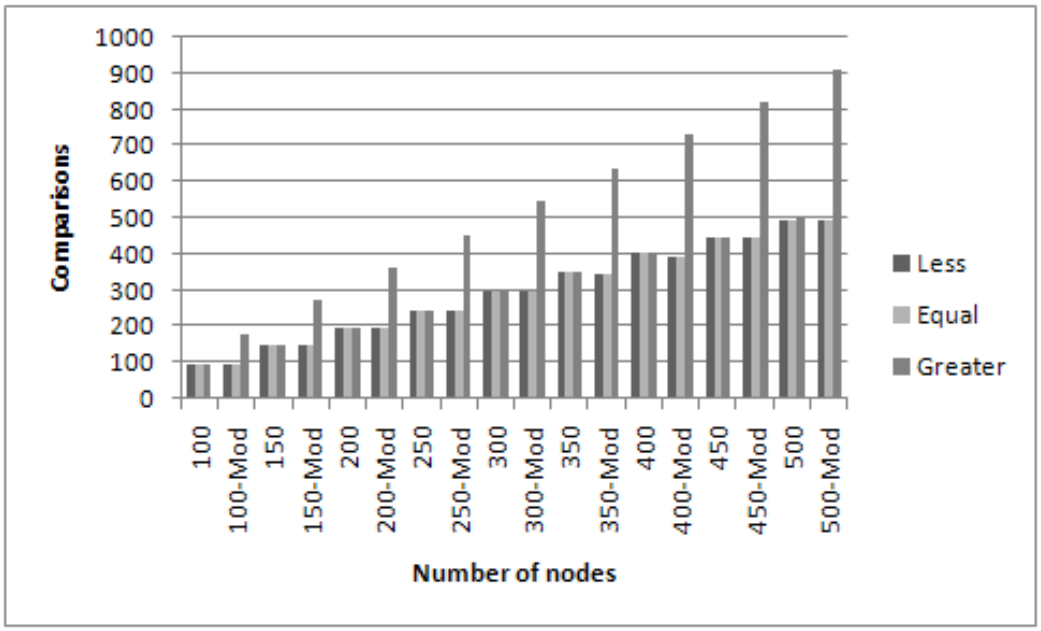


Figure 3.31: Decrease-key methodically on 2-3 heap

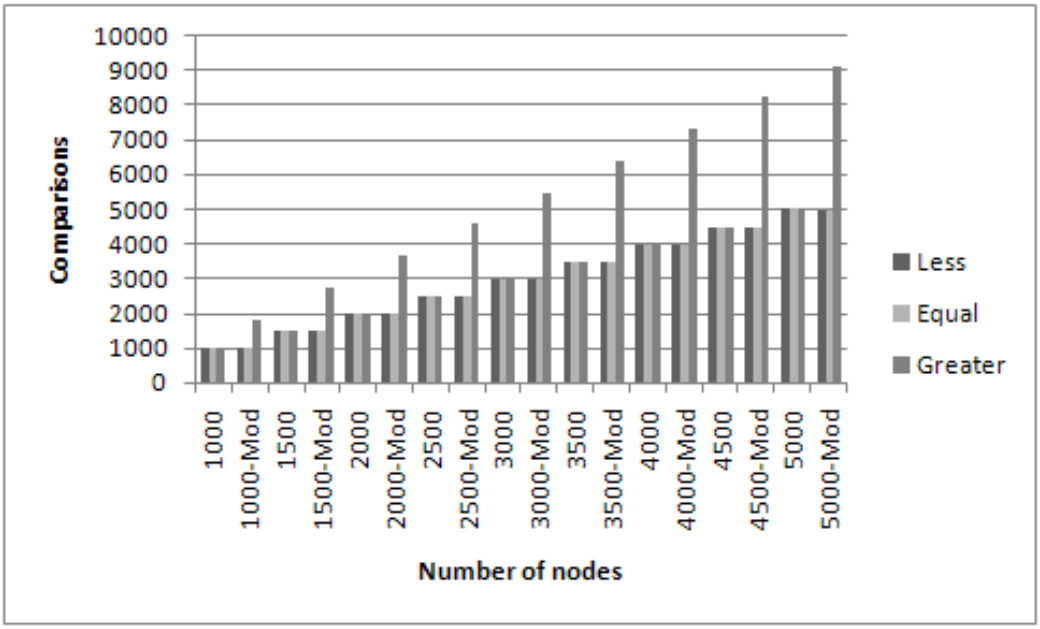


Figure 3.32: Decrease-key methodically on 2-3 heap

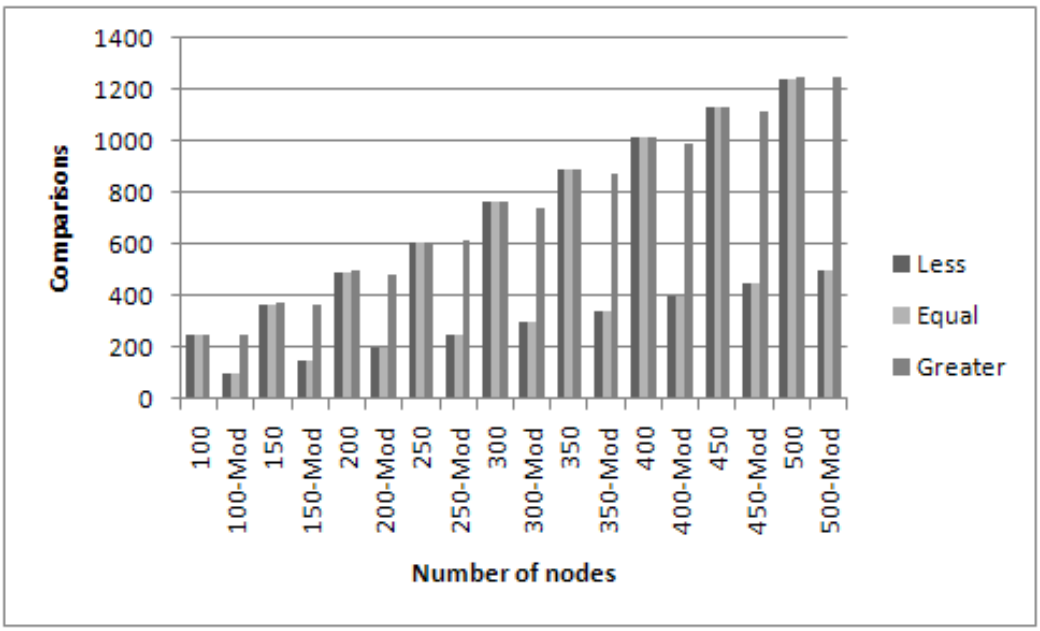


Figure 3.33: Decrease-key methodically on 3-4 heap

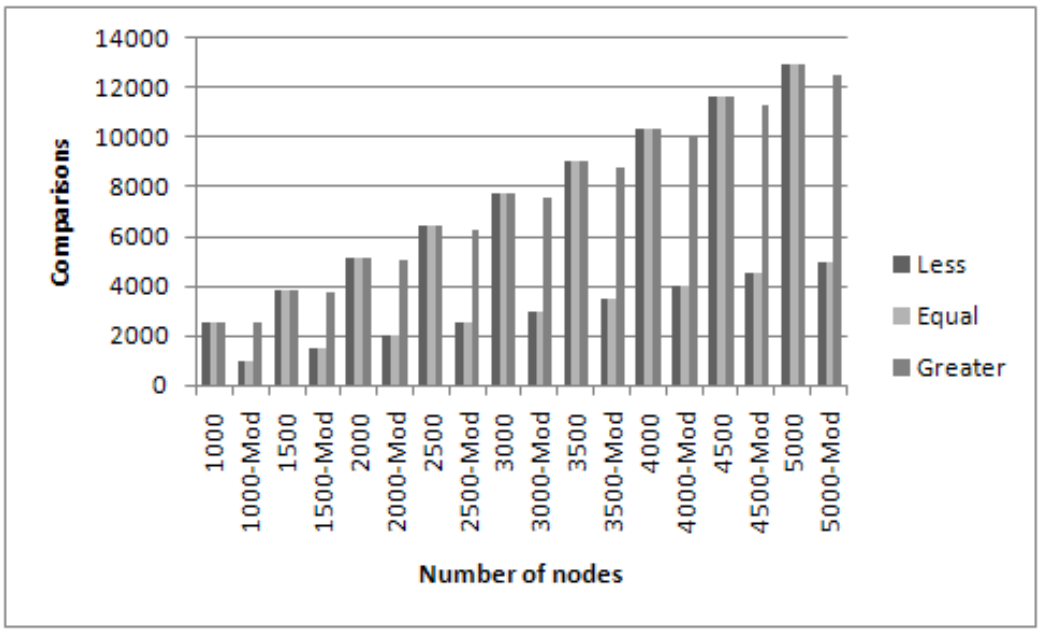


Figure 3.34: Decrease-key methodically on 3-4 heap

| Number | '$n$' to first | | First to '$n$' | | Random | |
|---|---|---|---|---|---|---|
| of Nodes | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap |
| 100 | 97 | 246 | 156 | 267 | 133 | 213 |
| 150 | 147 | 370 | 237 | 390 | 204 | 330 |
| 200 | 198 | 495 | 311 | 504 | 285 | 453 |
| 250 | 247 | 610 | 388 | 660 | 359 | 543 |
| 300 | 298 | 763 | 479 | 798 | 441 | 684 |
| 350 | 350 | 891 | 560 | 921 | 496 | 786 |
| 400 | 404 | 1019 | 639 | 1086 | 569 | 931 |
| 450 | 446 | 1132 | 721 | 1169 | 629 | 1024 |
| 500 | 497 | 1241 | 800 | 1322 | 708 | 1136 |
| 1000 | 994 | 2522 | 1602 | 2632 | 1438 | 2272 |
| 1500 | 1495 | 3864 | 2416 | 3941 | 2178 | 3478 |
| 2000 | 1999 | 5089 | 3207 | 5266 | 2877 | 4624 |
| 2500 | 2496 | 6431 | 4013 | 6628 | 3621 | 5770 |
| 3000 | 2996 | 7772 | 4802 | 7923 | 4326 | 6910 |
| 3500 | 3501 | 9061 | 5610 | 9250 | 5069 | 8095 |
| 4000 | 3995 | 10347 | 6387 | 10632 | 5818 | 9166 |
| 4500 | 4494 | 11635 | 7214 | 11900 | 6510 | 10370 |
| 5000 | 5001 | 12929 | 8015 | 13306 | 7226 | 11636 |

Table 3.4: Decrease-key key comparison costs for Figures 3.25–3.30

| Number | Standard Heap | | | Modified Heap | | |
|---|---|---|---|---|---|---|
| of Nodes | Less | Equal | Greater | Less | Equal | Greater |
| 100 | 97 | 97 | 97 | 97 | 97 | 179 |
| 150 | 147 | 147 | 147 | 146 | 146 | 273 |
| 200 | 198 | 198 | 198 | 196 | 196 | 364 |
| 250 | 247 | 247 | 247 | 247 | 247 | 453 |
| 300 | 298 | 298 | 298 | 297 | 297 | 547 |
| 350 | 350 | 350 | 351 | 345 | 345 | 639 |
| 400 | 404 | 404 | 404 | 394 | 394 | 730 |
| 450 | 446 | 446 | 446 | 446 | 446 | 823 |
| 500 | 497 | 497 | 498 | 496 | 496 | 912 |
| 1000 | 994 | 994 | 994 | 996 | 996 | 1828 |
| 1500 | 1495 | 1495 | 1495 | 1496 | 1496 | 2743 |
| 2000 | 1999 | 1999 | 2000 | 1996 | 1996 | 3663 |
| 2500 | 2496 | 2496 | 2496 | 2494 | 2494 | 4579 |
| 3000 | 2996 | 2996 | 2995 | 2996 | 2996 | 5496 |
| 3500 | 3501 | 3501 | 3502 | 3493 | 3493 | 6416 |
| 4000 | 3995 | 3995 | 3995 | 3993 | 3993 | 7328 |
| 4500 | 4494 | 4494 | 4494 | 4496 | 4496 | 8243 |
| 5000 | 5001 | 5001 | 5002 | 4994 | 4994 | 9161 |

Table 3.5: 2-3 heap decrease-key key comparison costs for Figures 3.31–3.32

| Number | Standard Heap | | | Modified Heap | | |
|---|---|---|---|---|---|---|
| of Nodes | Less | Equal | Greater | Less | Equal | Greater |
| 100 | 246 | 246 | 246 | 97 | 97 | 246 |
| 150 | 370 | 370 | 373 | 146 | 146 | 365 |
| 200 | 495 | 495 | 496 | 198 | 198 | 487 |
| 250 | 610 | 610 | 611 | 246 | 246 | 620 |
| 300 | 763 | 763 | 763 | 297 | 297 | 742 |
| 350 | 891 | 891 | 892 | 345 | 345 | 875 |
| 400 | 1019 | 1019 | 1019 | 397 | 397 | 992 |
| 450 | 1132 | 1132 | 1133 | 447 | 447 | 1115 |
| 500 | 1241 | 1241 | 1251 | 496 | 496 | 1247 |
| 1000 | 2522 | 2522 | 2523 | 996 | 996 | 2495 |
| 1500 | 3864 | 3864 | 3864 | 1495 | 1495 | 3756 |
| 2000 | 5089 | 5089 | 5089 | 1996 | 1996 | 5005 |
| 2500 | 6431 | 6431 | 6431 | 2496 | 2496 | 6261 |
| 3000 | 7772 | 7772 | 7773 | 2995 | 2995 | 7524 |
| 3500 | 9061 | 9061 | 9061 | 3495 | 3495 | 8785 |
| 4000 | 10347 | 10347 | 10348 | 3996 | 3996 | 10038 |
| 4500 | 11635 | 11635 | 11635 | 4495 | 4495 | 11276 |
| 5000 | 12929 | 12929 | 12930 | 4996 | 4996 | 12542 |

Table 3.6: 3-4 heap decrease-key key comparison costs for Figures 3.33–3.34

### *3.5.5 Conclusion*

Results generated by both heaps, standard and modified, display a linear complexity of $O(m)$.

For experiments run without the decrease-key modification, it can be seen that the 3-4 heap key comparison cost was always higher than the 2-3 heap. Whilst being higher, the absolute spread between the key comparison costs incurred for experiments a) through to c), was smallest. The absolute spread was calculated for both heaps by taking their highest and lowest key comparison cost for the aforementioned experiments, and comparing their respective spread size against each other. For experiment d), the key comparison costs incurred were essentially identical for both heaps respectively when the decrease-key operations involved decrements by a value which was 'less than', 'equal to', or 'greater than' the gap of two (2) with its immediate neighbour above.

Experiment d) was the only experiment performed with the decrease-key modification and it was on the 3-4 heap where the reduction in key comparisons occurred. For decrease-key values one (1) and two (2), the key comparison cost was half when compared to identical experiments performed without this modification. For decrease-key value three (3) the key comparison cost was less than without this modification because sufficient nodes remained in position due to tree restructuring and effectively cancelled the modification key comparison over head for when the nodes had to be removed by the standard process. Unlike for the 2-3 heap, for decrease-key values one (1) and two (2), the key comparison cost was essentially unchanged when compared to identical experiments performed without this modification. For decrease-key value three (3), the key comparison cost increased by approximately eighty five (85) percent.

## *3.6 Experiments with Single Source Shortest Path*

Continuing on from the previously detailed insert, delete-min and decrease-key experiments, these experiments investigate the complete picture of the 3-4 heap as an alternative to the 2-3 heap and will concentrate on both heaps being used as the data storage provider for Dijkstra's [2] 'Single Source

Shortest Path' algorithm. Of key interest is how the 3-4 heap will perform overall compared against the 2-3 heap because the key comparison cost for insert is worse, its delete-min is better and decrease-key is worse.

There are three distinct types of dataset modelling used where the vertices on the graph were respectively densely connected, sparsely connected and densely connected with osculating distances. The definition of a dense graph [8] is "a graph in which the number of edges is close to the possible number of edges", and a sparse graph [8] is "a graph in which the number of edges is much less than the possible number of edges".

The density of a graph is defined by the number of outgoing edges from each vertex. For the dense graph, this is defined by the probability of an edges existence $p$ where the total number of edges, $m$, is given by $m = pn^2$, where $n$ is the number of vertices. For the sparse graph, this is defined by an edge factor which is the average number of outgoing edges from each vertex, where the total number of edges, $m$, is given by $m = fn$, where $n$ is the number of vertices.

For the dense graph, the probability parameter of an edge existence is $p = 1.0$, thus there will be about $(1.0)n^2$ edges in total which means there will be about $n$ edges for each vertex. For the sparse graph, the outgoing edge factor is $f = 4.0$, thus there will be about $(4.0)n$ edges in total which means will be about (4.0) edges for each vertex. Using the sparse graph Dijkstra complexity is $O(m + n \log n) = O(n \log n)$ and since $m = O(fn) = O(n)$, this is absorbed into $O(n \log n)$. Using the dense graph Dijkstra complexity is $O(m+n \log n) = O(n^2)$ and since $m = O(pn^2)$, this will dominate $O(n \log n)$.

A dense graph with osculating distances is a dense graph where the values associated with each edge have been generated in a series. The pattern used for this series has the seed value incremented by one (1) consecutively three times before a new series seed was randomly generated with modulus five hundred (500). The dense graph with osculating distances was generated to discover if under ideal operating conditions as a data store, the 3-4 heap performance would be better than the 2-3 heap. This type of dataset would make the 3-4 heap perform better for both insert and delete-min operations so it would be interesting to see if these would do more than counter balance the worse performing decrease-key and allow the 3-4 heap to have the lowest

key comparison cost.

For each experiment, the number of vertices in the data set graph range from one hundred (100) to five thousand (5,000). Graph sizes between one hundred (100) to five hundred (500) vertices used an incremental size of fifty (50), and graph sizes between one thousand (1000) to five thousand (5000) vertices used an incremental size of five hundred (500). The letter '$n$' is used to represent the total number of vertices contained in each dataset during an experiment. Each experiment was also performed twice, once with the heaps as standard and once with the insert cache and decrease-key modifications enabled. Both modifications are enabled because the coverage of these experiments utilise all of the heaps core functionality comprising of insert, delete-min and decrease-key. Only one set of experiments was conducted for dense, sparse and osculating dense graphs, versus a more methodical approach as in Sections 3.2–3.5. This was done because these experiments presented a clear profile of the 3-4 heap performance as a data store for Dijkstra's [2] 'Single Source Shortest Path' algorithm.

One threat to experiment validity is having identical experiments being performed on each data structure using a slightly different graph data set. To ensure that this does not occur, both data structures will be operated in series with the same graph data set.

### *3.6.1 Dense Graph*

Results generated by both heaps, standard and modified, display the complexity of $O(m + n \log n)$, where $m = pn^2$.

For experiments run without the insert cache and decrease-key modification, it can be seen that the 3-4 heap key comparison cost was always higher than the 2-3 heap.

For experiments run with the insert cache and decrease-key modification, it can be seen that the 3-4 heap key comparison cost was always higher than the 2-3 heap. Whilst being higher, the difference in key comparisons between both heaps was smaller than when compared against results from without modification. This is because the key comparison cost incurred by the 2-3 heap increased due to the insert cache modification, whilst the key compari-

son cost decreased for the 3-4 heap due to the decrease-key modification.

Inspecting the difference in key comparisons incurred during the modified and unmodified experiments, it can be seen that both heaps are slowly converging together. It was expected that once the modifications were enabled, the 3-4 heap key comparison costs would become lower than the 2-3 heap because the 2-3 heap key comparison cost would increase and the 3-4 heap decrease. Whilst this conversion of key comparison costs did eventuate, it was not sufficiently large enough to give 3-4 heap the lowest key comparison cost.

Experiment results data used to generate each figure presented in this section are shown in Table 3.7 on page 87.

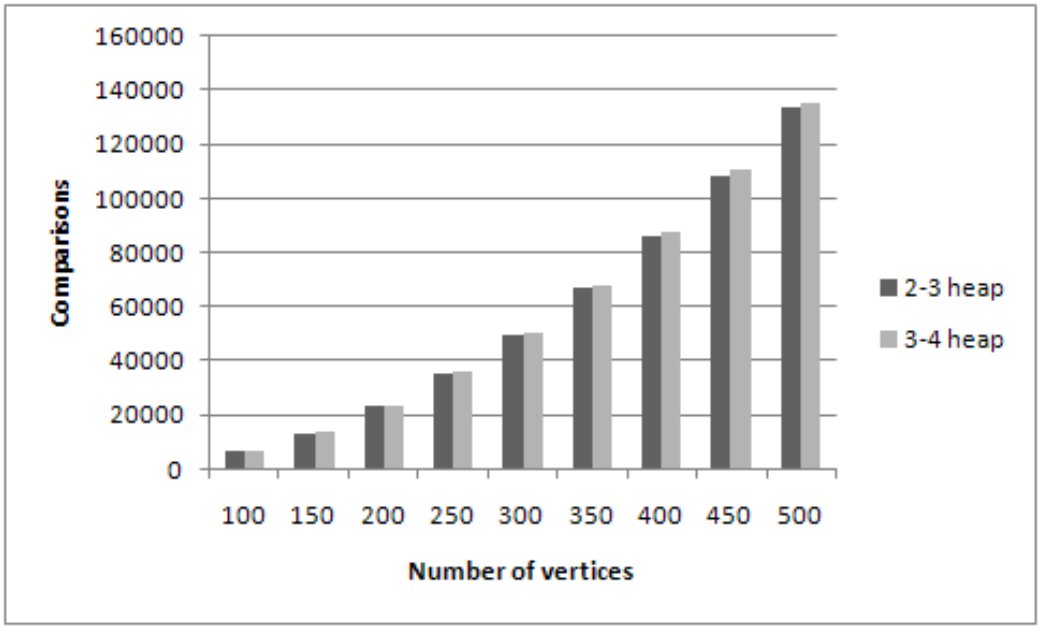


Figure 3.35: Dijkstra dense graph using standard 2-3 heap and 3-4 heap

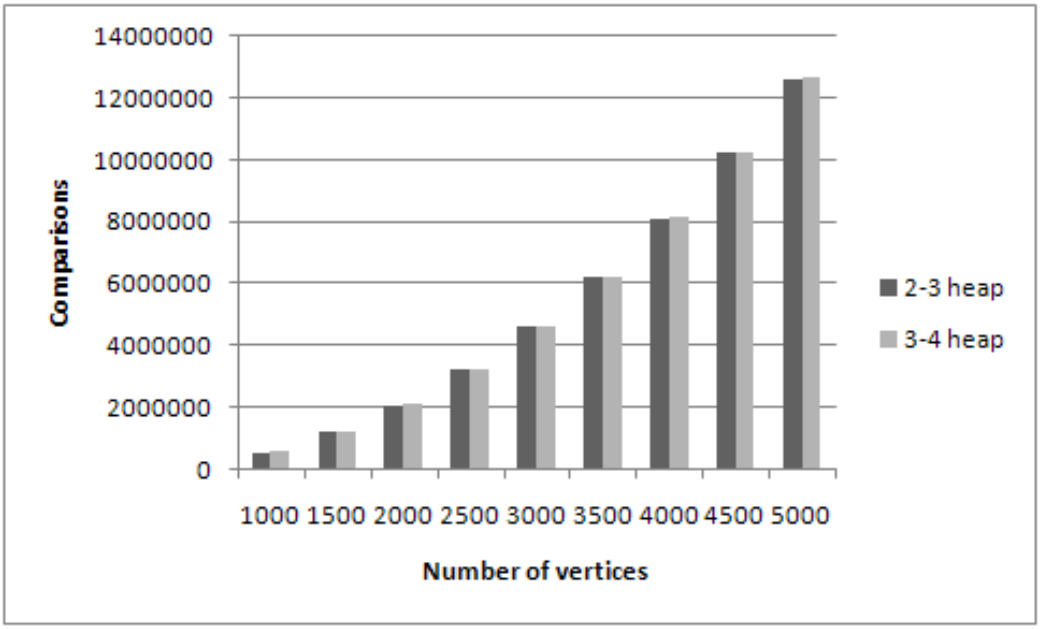


Figure 3.36: Dijkstra dense graph using standard 2-3 heap and 3-4 heap

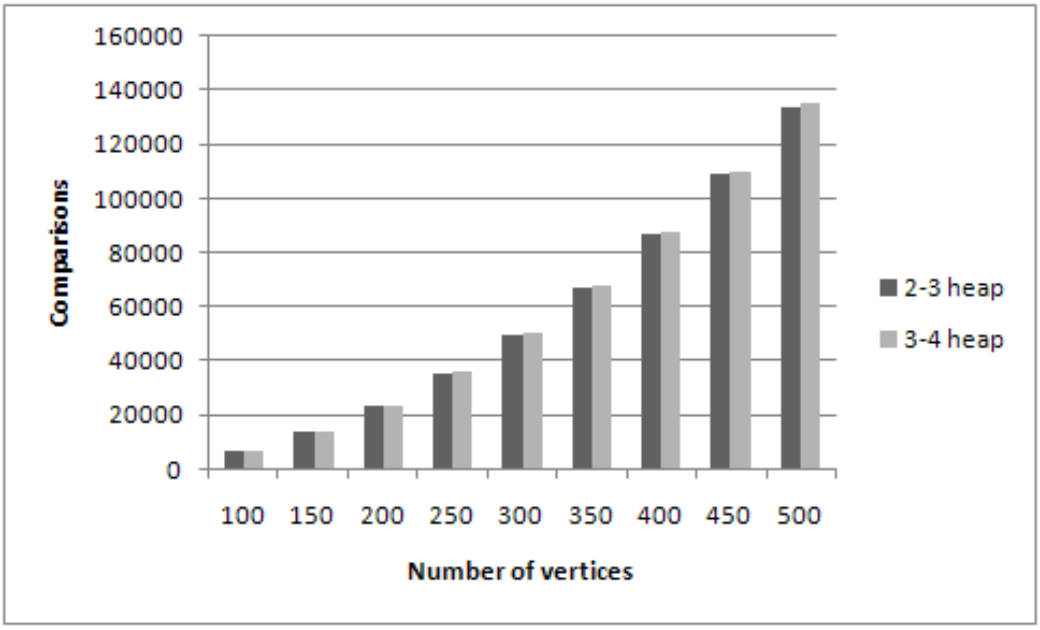


Figure 3.37: Dijkstra dense graph using modified 2-3 heap and 3-4 heap

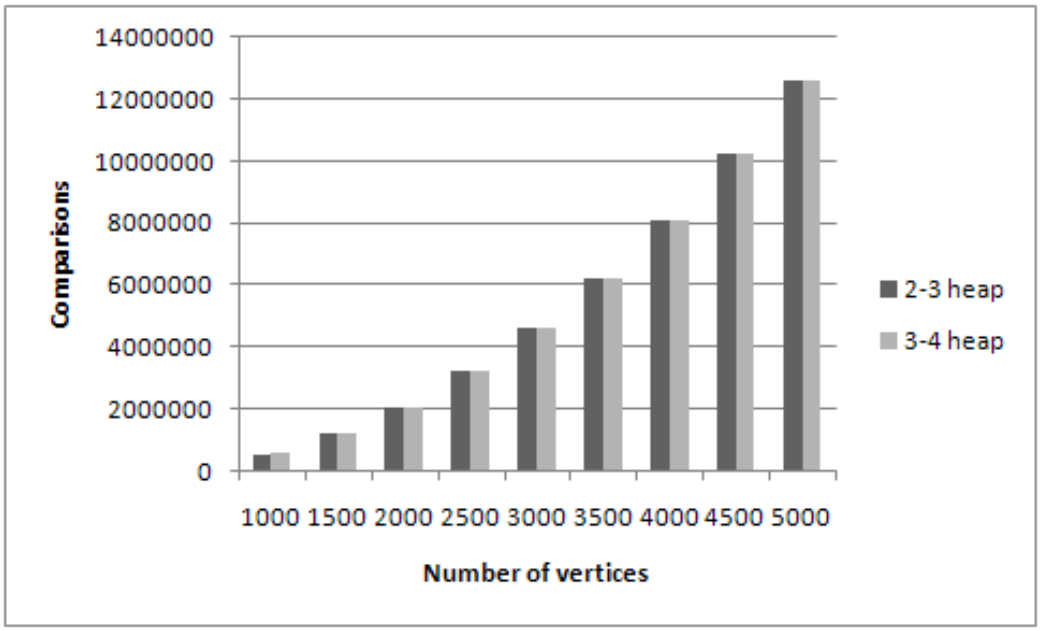


Figure 3.38: Dijkstra dense graph using modified 2-3 heap and 3-4 heap

### *3.6.2 Sparse Graph*

Results generated by both heaps, standard and modified, does not exactly display the complexity of $O(m + n \log n)$, but there is present an upwards sloping trend line which approximately follows the complexity of $O(m + n \log n)$, where $m = fn$.

For experiments run without the insert cache and decrease-key modification, it can be seen that the 3-4 heap key comparison cost was always, except for one, higher than the 2-3 heap for small values of '$n$' and was mostly lower than for the larger values of '$n$'.

For experiments run with the insert cache and decrease-key modification, it can be seen that the 3-4 heap key comparison cost was always, except for one, higher than the 2-3 heap for small values of '$n$' and always lower for large values of '$n$'. The key comparisons cost incurred by both heaps increased but the difference in key comparisons incurred between both heaps has remained more or less static, so there was no noticeable increase or decrease in key comparisons which can be attributed towards either modification.

Experiment results data used to generate each figure presented in this

section are shown in Table 3.7 on page 87.

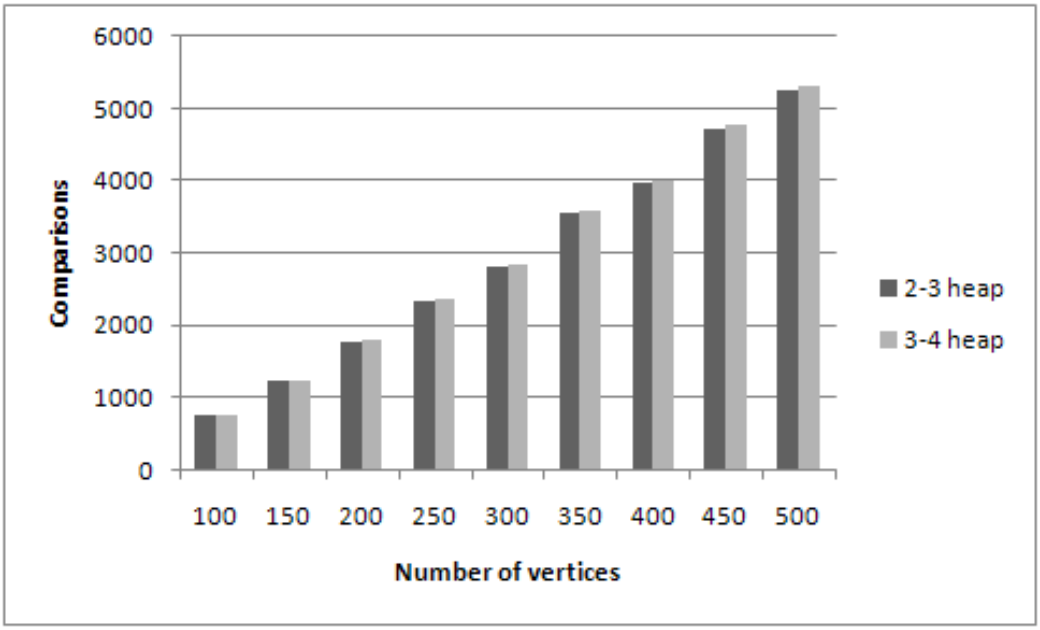


Figure 3.39: Dijkstra sparse graph using standard 2-3 heap and 3-4 heap

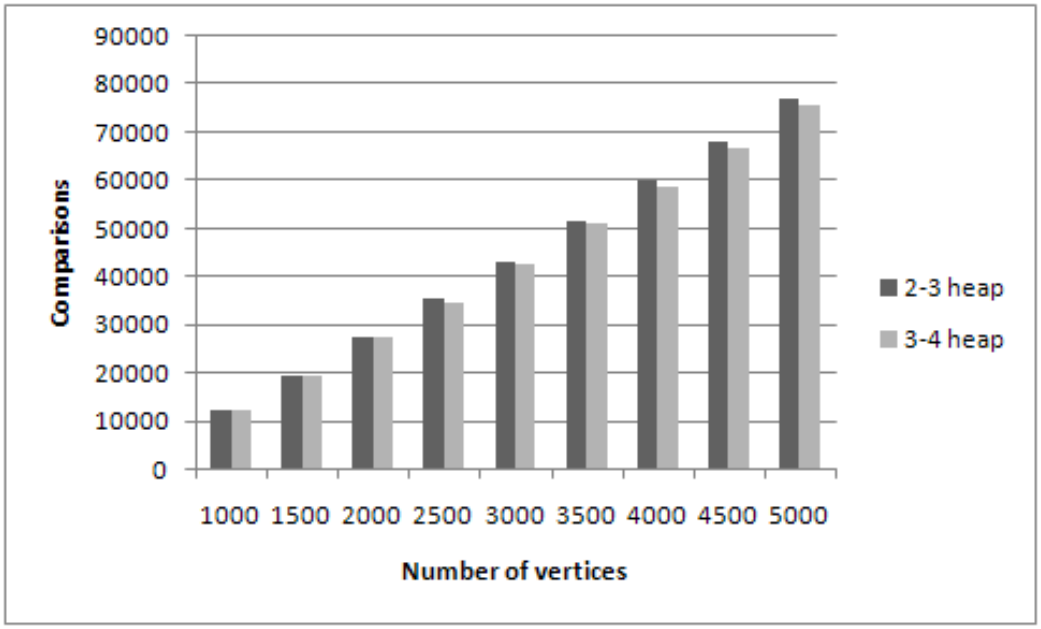


Figure 3.40: Dijkstra sparse graph using standard 2-3 heap and 3-4 heap

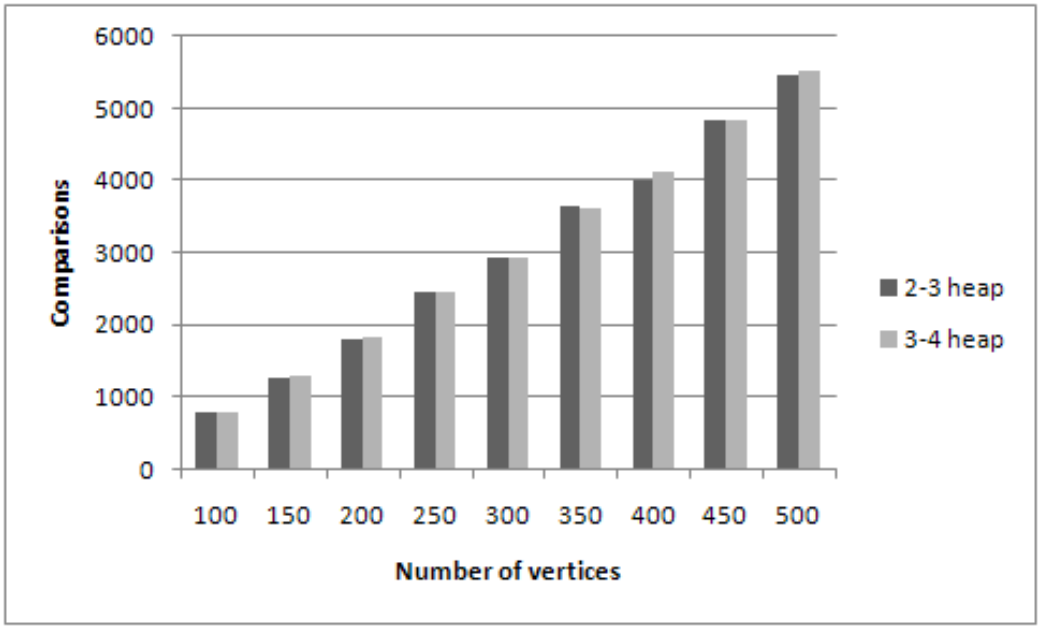


Figure 3.41: Dijkstra sparse graph using modified 2-3 heap and 3-4 heap

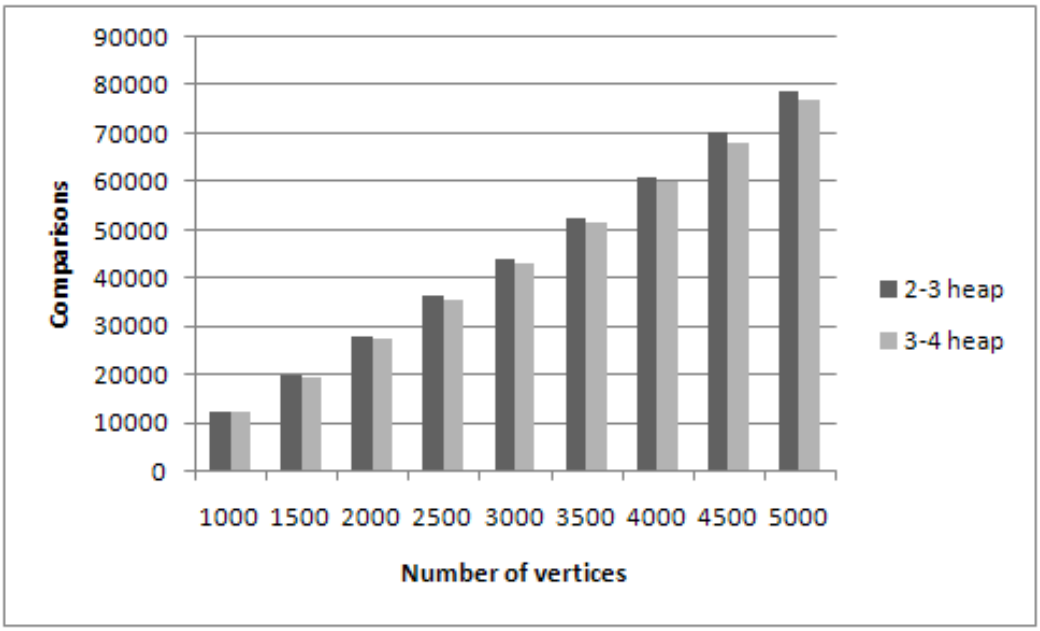


Figure 3.42: Dijkstra sparse graph using modified 2-3 heap and 3-4 heap

### *3.6.3 Osculating Dense Graph*

Results generated by both heaps, standard and modified, display the complexity of $O(m + n \log n)$, where $m = pn^2$.

For experiments run without the insert cache and decrease-key modification, it can be seen that the 3-4 heap key comparisons was higher than 2-3 heap for graphs with data set sizes five hundred (500) vertices and smaller, whilst lower than for graphs with data set sizes one thousand (1000) vertices and larger.

For experiments run with the insert cache and decrease-key modification, it can be seen that the 3-4 heap key comparison cost was always lower than the 2-3 heap. Whilst being lower for all graph data set sizes, the number of key comparisons incurred by the 2-3 heap had actually reduced in comparison to results without modification. The reduction of key comparisons incurred by the 2-3 heap was thirty (30) percent of the reduction achieved by the 3-4 heap for the same graph data set. Because the reduction amount was smaller, this allowed the 3-4 heap to always have the lowest key comparison cost.

Experiment results data used to generate each figure presented in this section are shown in Table 3.8 on page 88.

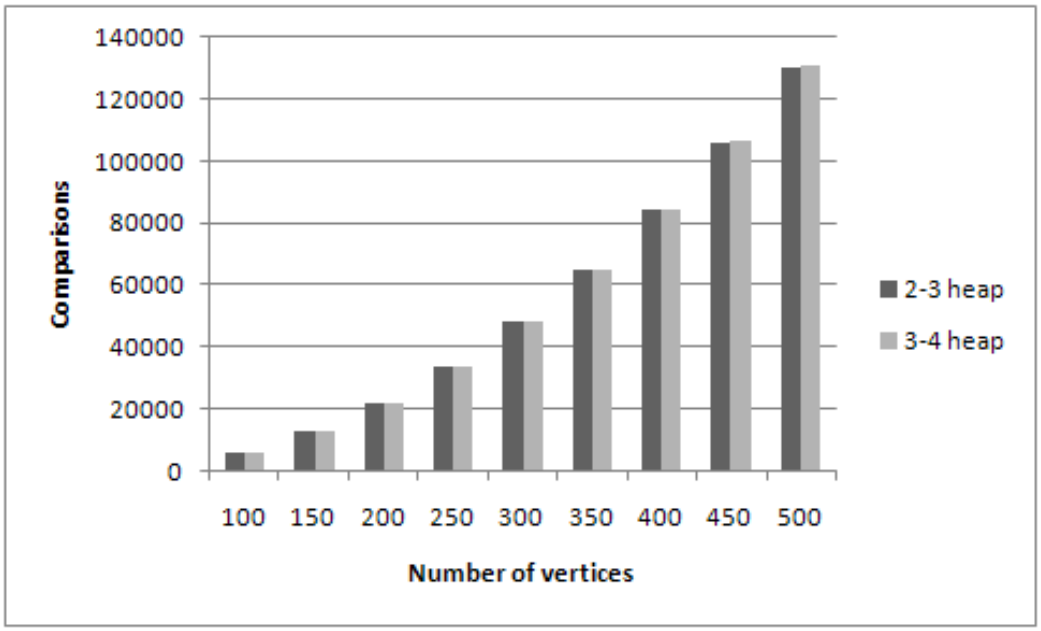


Figure 3.43: Dijkstra osculating dense graph using standard 2-3 heap and 3-4 heap

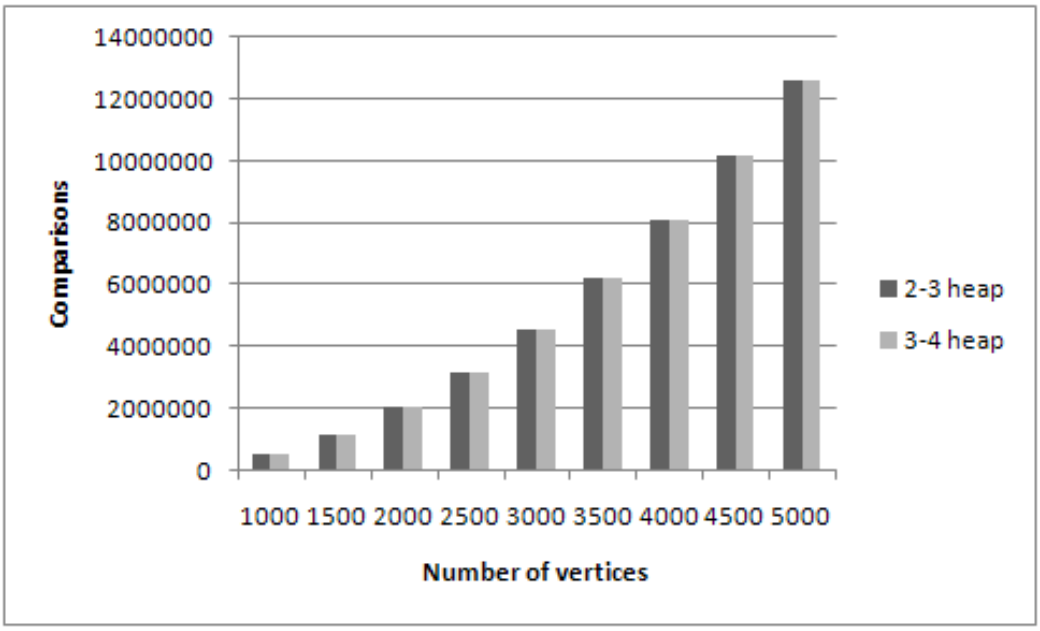


Figure 3.44: Dijkstra osculating dense graph using standard 2-3 heap and 3-4 heap

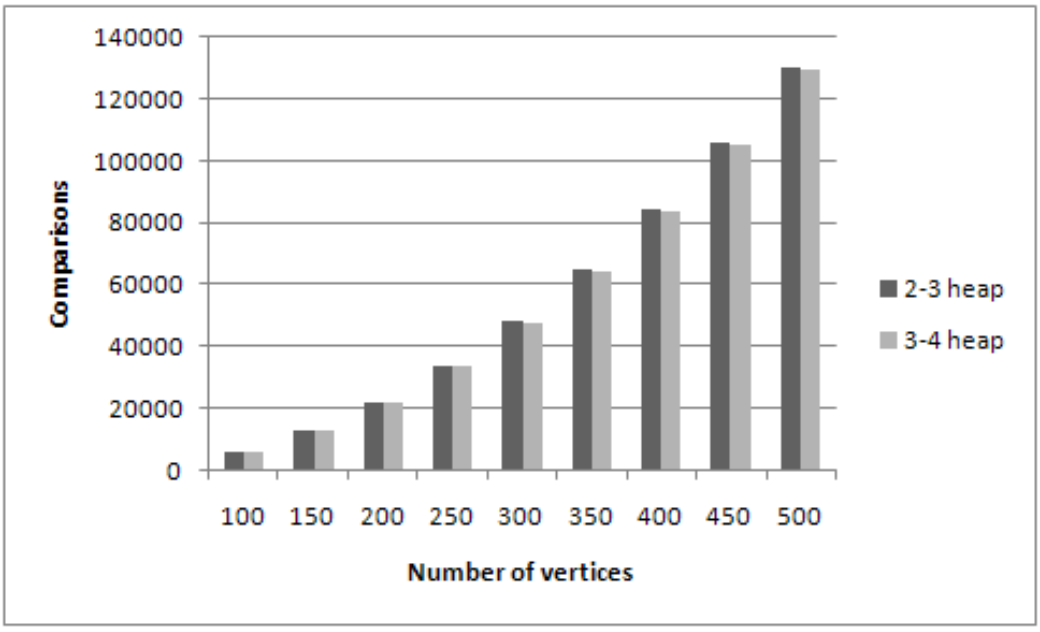


Figure 3.45: Dijkstra osculating dense graph using modified 2-3 heap and 3-4 heap

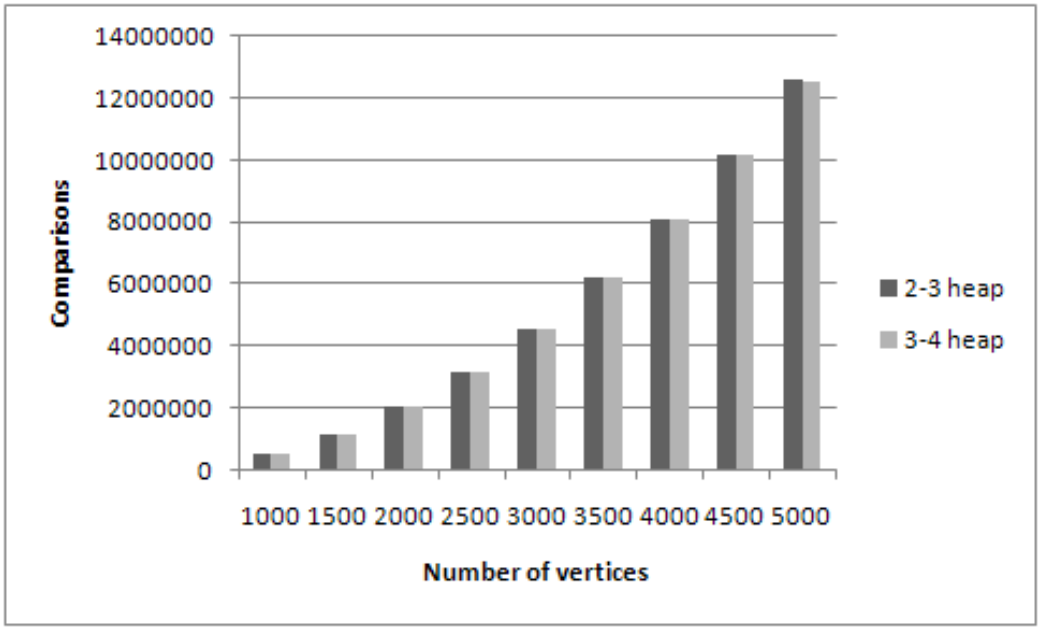


Figure 3.46: Dijkstra osculating dense graph using modified 2-3 heap and 3-4 heap

| | Dense Graph | | | | Sparse Graph | | | |
|---|---|---|---|---|---|---|---|---|
| Number | Standard | | Modified | | Standard | | Modified | |
| of Vertices | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap |
| 100 | 6002 | 6319 | 6053 | 6230 | 746 | 752 | 768 | 778 |
| 150 | 13027 | 13650 | 13178 | 13532 | 1222 | 1211 | 1240 | 1277 |
| 200 | 22712 | 23386 | 22779 | 23258 | 1771 | 1792 | 1795 | 1822 |
| 250 | 34768 | 35687 | 34983 | 35488 | 2328 | 2357 | 2434 | 2452 |
| 300 | 49225 | 50495 | 49531 | 50283 | 2820 | 2821 | 2923 | 2937 |
| 350 | 66584 | 67846 | 66768 | 67521 | 3544 | 3589 | 3651 | 3613 |
| 400 | 86236 | 87867 | 86509 | 87453 | 3967 | 3987 | 4003 | 4117 |
| 450 | 108586 | 110402 | 108792 | 109958 | 4703 | 4778 | 4825 | 4839 |
| 500 | 133376 | 135508 | 133702 | 135005 | 5240 | 5309 | 5452 | 5522 |
| 1000 | 519836 | 524488 | 520303 | 523121 | 11898 | 11910 | 12237 | 12186 |
| 1500 | 1157348 | 1165082 | 1159060 | 1162684 | 19211 | 19240 | 19673 | 19400 |
| 2000 | 2045895 | 2055400 | 2048031 | 2052514 | 27218 | 27145 | 27850 | 27385 |
| 2500 | 3184415 | 3196360 | 3186719 | 3191967 | 35248 | 34666 | 36115 | 35255 |
| 3000 | 4572285 | 4586539 | 4574755 | 4581050 | 42922 | 42652 | 44028 | 43200 |
| 3500 | 6210826 | 6227729 | 6213697 | 6220258 | 51299 | 50987 | 52337 | 51641 |
| 4000 | 8099458 | 8117915 | 8102086 | 8110260 | 59903 | 58811 | 61114 | 59874 |
| 4500 | 10237546 | 10258073 | 10240340 | 10248992 | 68136 | 66816 | 70198 | 68153 |
| 5000 | 12627462 | 12650189 | 12630175 | 12640455 | 77206 | 75698 | 78591 | 76846 |

Table 3.7: Dijkstra osculating dense graph key comparisons for Figures 3.35–3.42

| Number of Vertices | Osculating Dense Graph | | | |
|---|---|---|---|---|
| | Standard | | Modified | |
| | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap |
| 100 | 5727 | 5811 | 5672 | 5630 |
| 150 | 12503 | 12599 | 12415 | 12333 |
| 200 | 21777 | 21896 | 21689 | 21558 |
| 250 | 33599 | 33739 | 33487 | 33301 |
| 300 | 47986 | 48106 | 47836 | 47587 |
| 350 | 64848 | 65013 | 64684 | 64361 |
| 400 | 84248 | 84379 | 84047 | 83707 |
| 450 | 106156 | 106308 | 105923 | 105496 |
| 500 | 130562 | 130687 | 130367 | 129821 |
| 1000 | 512855 | 512754 | 512333 | 511009 |
| 1500 | 1145672 | 1145469 | 1145034 | 1142774 |
| 2000 | 2029127 | 2028406 | 2027890 | 2024769 |
| 2500 | 3162699 | 3161656 | 3161340 | 3157013 |
| 3000 | 4546553 | 4544967 | 4544996 | 4539676 |
| 3500 | 6180524 | 6178390 | 6178830 | 6172257 |
| 4000 | 8064780 | 8062232 | 8062797 | 8055169 |
| 4500 | 10199156 | 10196064 | 10196904 | 10188169 |
| 5000 | 12583692 | 12580137 | 12581024 | 12571144 |

Table 3.8: Dijkstra osculating dense graph key comparisons for Figures 3.43–3.46

### *3.6.4 Conclusion*

Results generated by both heaps when standard and modified for the dense and sparse graphs, display the complexity of $O(m + n \log n)$, where respectively $m$ is defined as $pn^2$ and $fn$.

For experiments run without the insert cache and decrease-key modification, as the value of '$n$' became larger, the key comparison cost differences between the 2-3 heap and 3-4 heap converged together in experiments performed using sparse graph and dense graph with osculating distances. For some experiments, the 3-4 heap managed to have a lower key comparison cost.

For experiments run with the insert cache and decrease-key modification, as the value of '$n$' became larger, the key comparison cost differences between the 2-3 heap and 3-4 heap converged together and ultimately resulted in the 3-4 heap having the lowest for experiments performed using the sparse graph. For experiments performed using the dense graph with osculating distances, the 3-4 heap always had the lowest key comparison cost. For experiments performed using the dense graph, the 3-4 heap always had the highest key comparison cost.

The overall impact of the insert cache and decrease-key modifications is different for each of the heaps. The key comparison cost incurred by the 2-3 heap would increase slightly whilst the 3-4 heap would experience a slight reduction. This results in a convergence of key comparisons when comparing modified versus unmodified results. This convergence also increases the possibility for the 3-4 heap to have the lowest key comparison cost for an experiment. In the case of experiments performed using the graph with osculating distances, the 3-4 heap always had the lowest key comparison cost and this was despite the 2-3 heap also experiencing a reduction in key comparisons.

## *3.7 Measuring CPU Time Complexity*

This section presents the CPU time complexity, abbreviated as time complexity, required by the 3-4 heap in comparison to the 2-3 heap for insert, delete-min, decrease-key and as the data store for Dijkstra's [2] 'Single Source

Shortest Path' algorithm. Not all previously run experiments had their time complexity measured, instead only one scenario from each section was selected. Time complexity is measured in milliseconds and since the times slices measured were sometimes very small, any spikes in the results with small durations were mostly likely caused by external environmental factors such as CPU time slicing and hard disk access. These spikes are therefore excluded from analysis such that results analysis remains focused on the general time complexity line. It is expected that the time complexity results will mirror the key comparisons cost results, so for key comparison experiments where the 3-4 heap incurred the highest, the time complexity measurements would mirror this.

The computer equipment specifications used to generate these results is: Intel(R) Core(TM)2 Duo E6750 @ 2.66GHz with 2GB RAM running operating system Fedora Core 8 with kernel 2.6.23.14-155.fc8 and compiler GCC version 4.12 20070925 with compile options '-Wall -0'. All experiment results in this section were produced 'warm' whereby the first set of results generated were discarded and the second set of results generated were taken as the actual experiment results. Using 'warm' results was done to eliminate any execution pauses while the hardware and software prepared sufficient CPU cache and ordinary system RAM and any operating system calls were also warmed up. By having the environment warmed up, this ensured execution wasn't paused and therefore the times generated were as accurate as possible. It also meant that the most frequently recently used memory locations were in the faster CPU cache.

Time complexity results for the core functional area of insert are presented in Figures 3.47–3.50 and were generated using the insert experiment with osculating numerical data. For experiments performed with both heaps being standard, the 3-4 heap incurred a higher time complexity. For experiments performed with both heaps being modified, with the exception of two experiments, the time complexity used by the 3-4 heap was lower than the 2-3 heap. These results mirror the results from the key comparison experiments and time complexity increased with a linear complexity of $O(m)$.

Time complexity results for the core functional area of delete-min are presented in Figures 3.51–3.52 and were generated using the delete-min with

decreasing numerical experiment. With the exception of one measurement, the time complexity used by the 3-4 heap was lower than the 2-3 heap. These results mirror the results from the key comparison experiments and time complexity increased with a complexity of $O(n \log\ n)$.

Time complexity results for the core functional area of decrease-key are presented in Figures 3.53–3.56 and were generated using the decrease-key methodically experiment. For experiments performed with both heaps being standard, the 3-4 heap incurred a higher time complexity. For experiments performed with both heaps being modified, the time complexity for 'less than' and 'equal to' where essentially identical between both heaps. Results for 'larger than' had all reduced in comparison with experiment results for when both heaps were standard. Excluding the reduction in time complexity for the modified 2-3 heap, these results mirror the results from the key comparison experiments and time complexity increased with a linear complexity of $O(m)$.

Time complexity results for Dijkstra's [2] 'Single Source Shortest Path' algorithm are presented in Figures 3.57–3.60 and were generated using the dense graph experiment. For experiments performed with both heaps being standard, the 3-4 heap incurred a higher time complexity. For experiments performed with both heaps being modified, time complexity increased for the 2-3 heap and reduced for the 3-4 heap, unfortunately not sufficiently for the 3-4 heap to have the lowest time complexity. These results mirror the results from the key comparison experiments and time complexity increased with a complexity of $O(m\ +\ n \log\ n)$, where $m = pn^2$.

### *3.7.1 Conclusion*

There is a direct relationship between the number of key comparisons incurred and time complexity, this applies to both the 2-3 heap and most importantly the 3-4 heap.

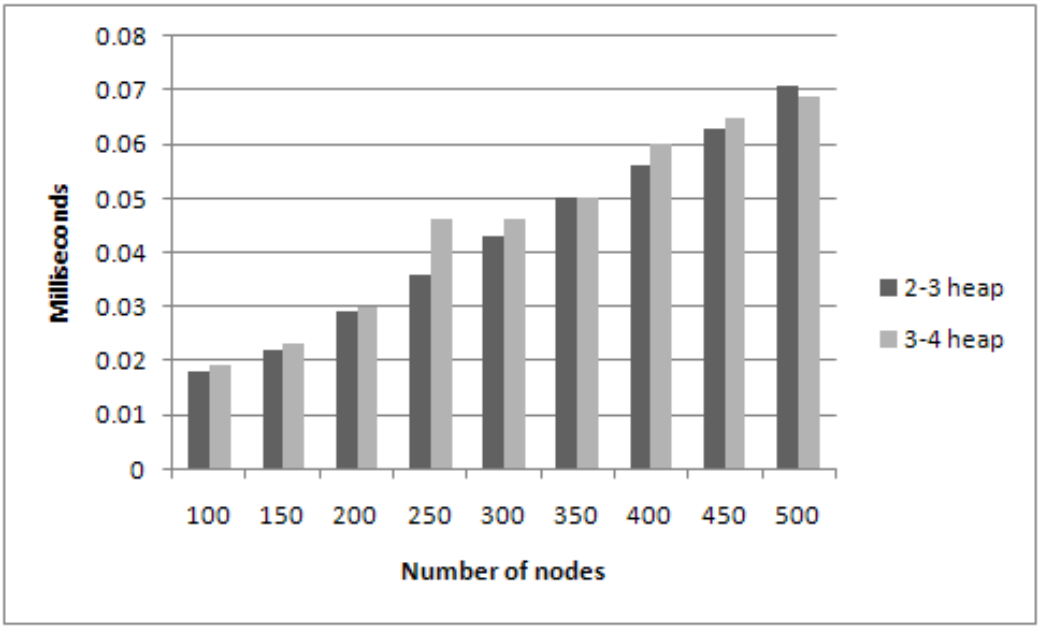


Figure 3.47: Insert time complexity using standard 2-3 heap and 3-4 heap

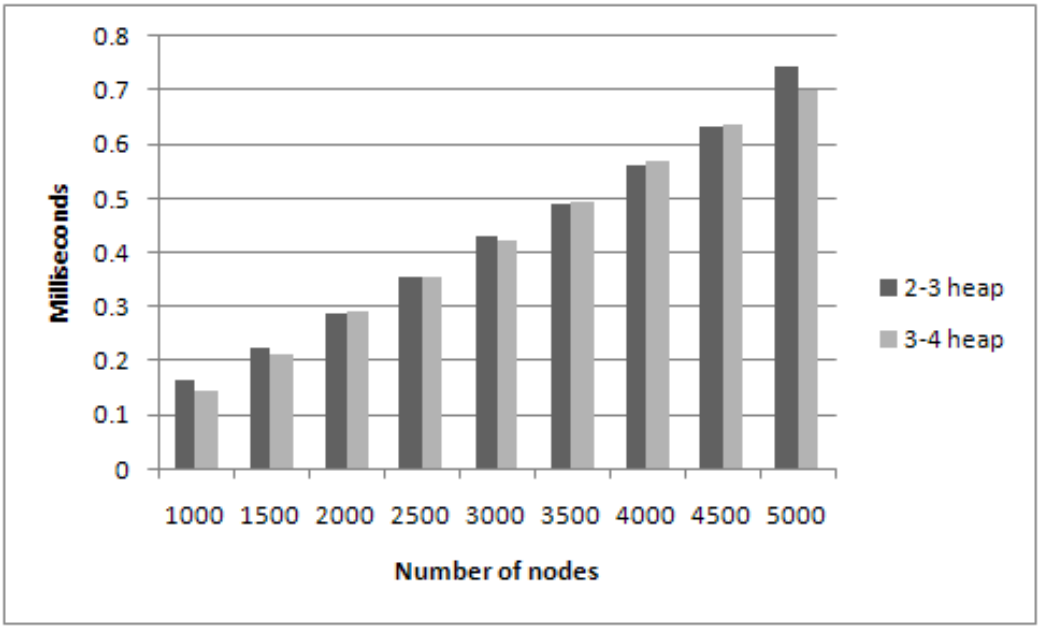


Figure 3.48: Insert time complexity using standard 2-3 heap and 3-4 heap

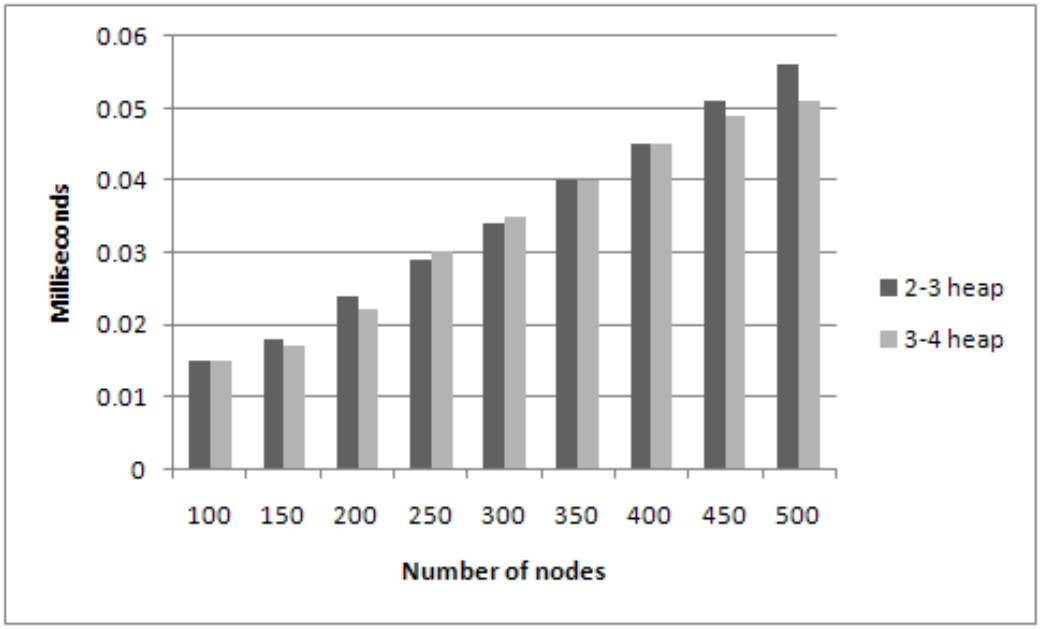


Figure 3.49: Insert time complexity using modified 2-3 heap and 3-4 heap

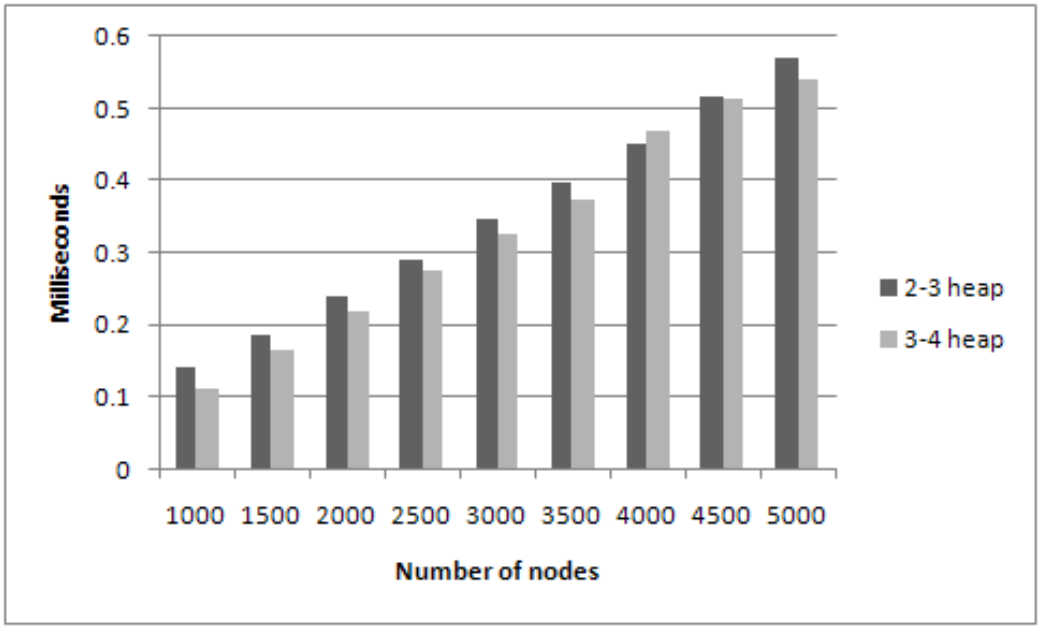


Figure 3.50: Insert time complexity using modified 2-3 heap and 3-4 heap

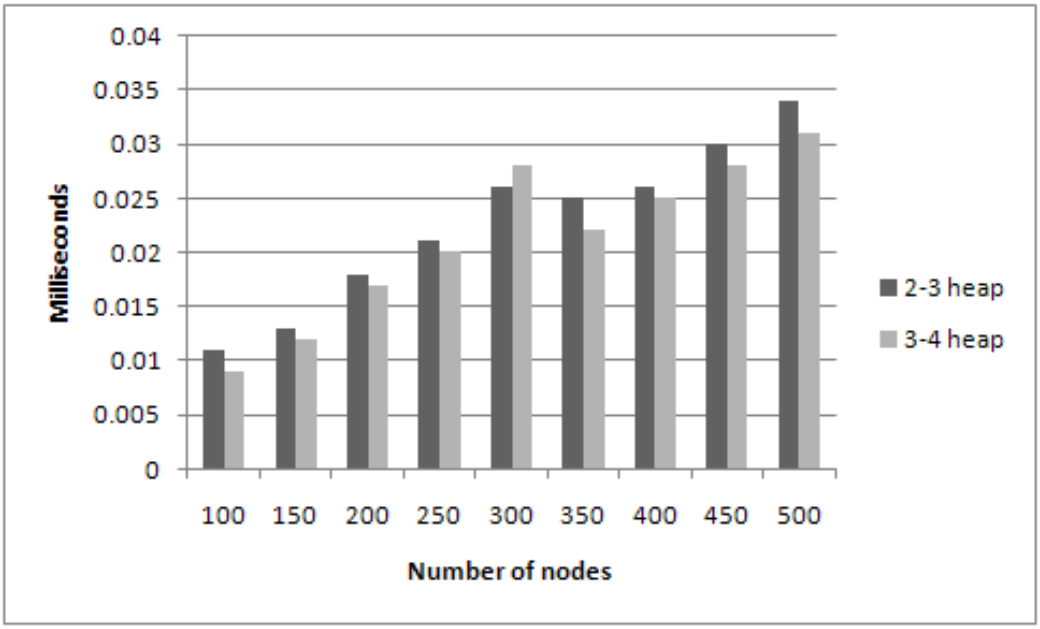


Figure 3.51: Delete-min time complexity 2-3 heap and 3-4 heap

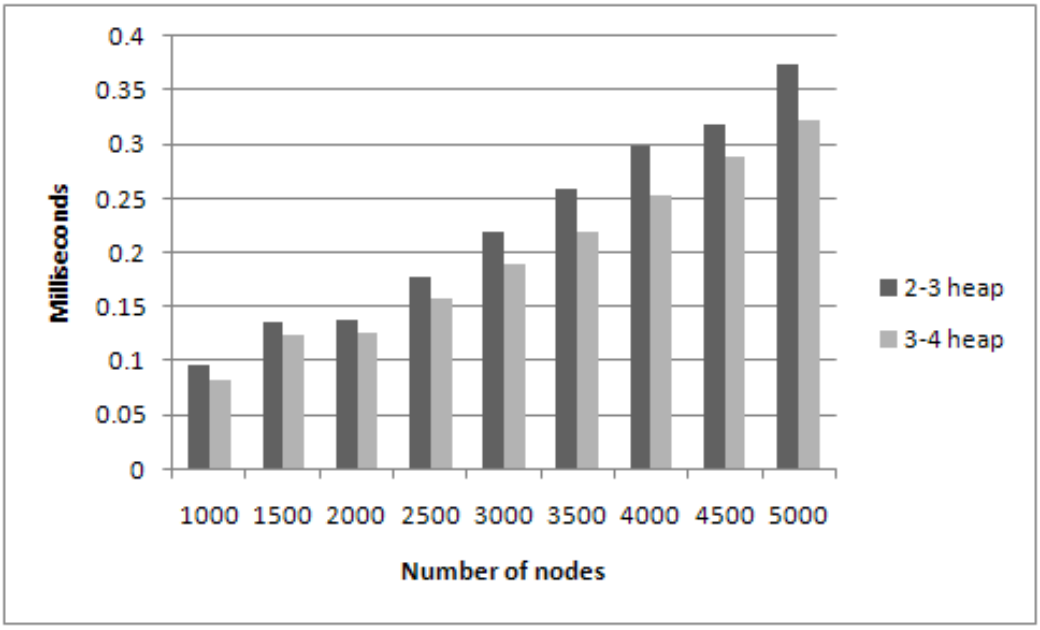


Figure 3.52: Delete-min time complexity 2-3 heap and 3-4 heap

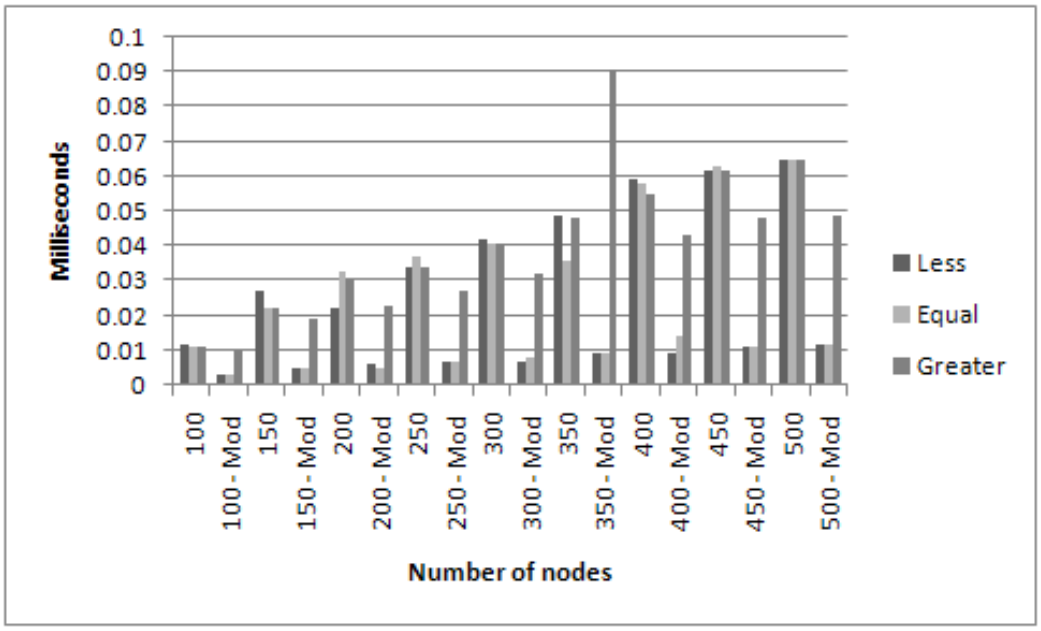


Figure 3.53: Decrease-key time complexity for standard and modified 2-3 heap

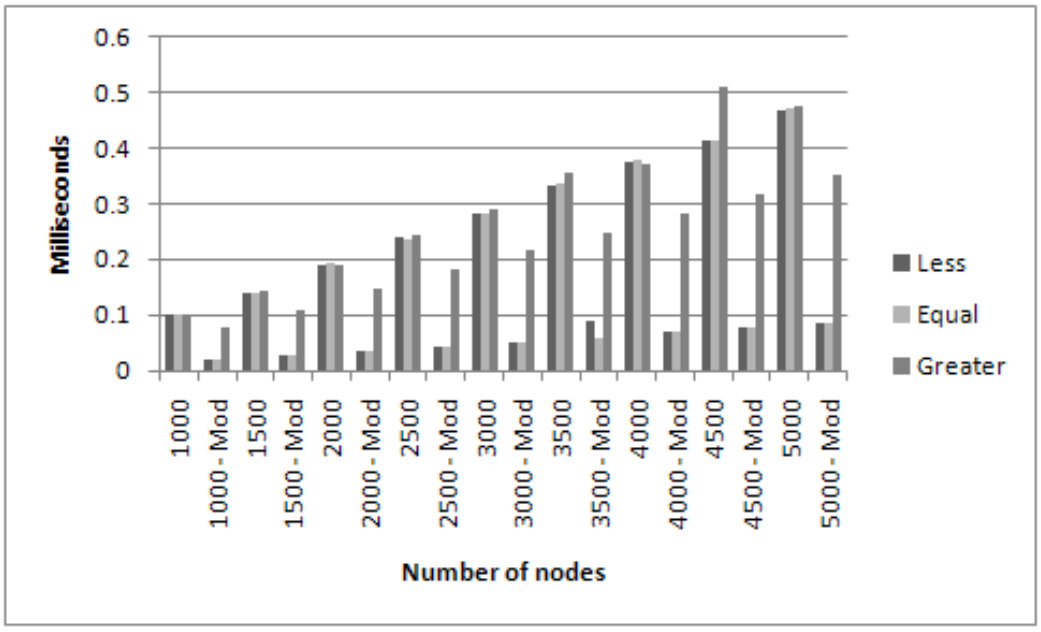


Figure 3.54: Decrease-key time complexity for standard and modified 2-3 heap

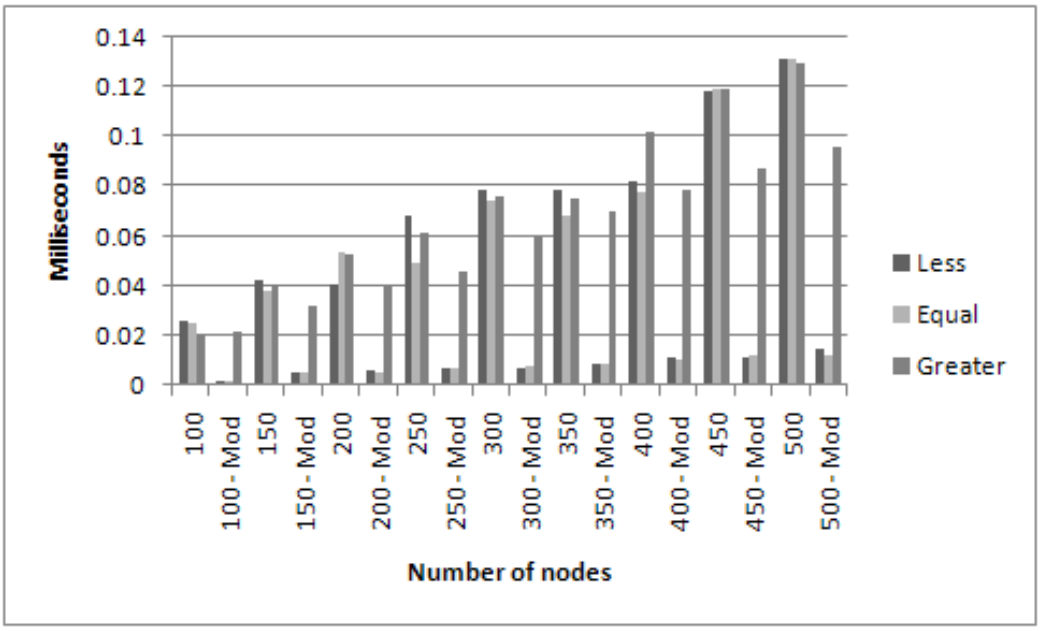


Figure 3.55: Decrease-key time complexity for standard and modified 3-4 heap

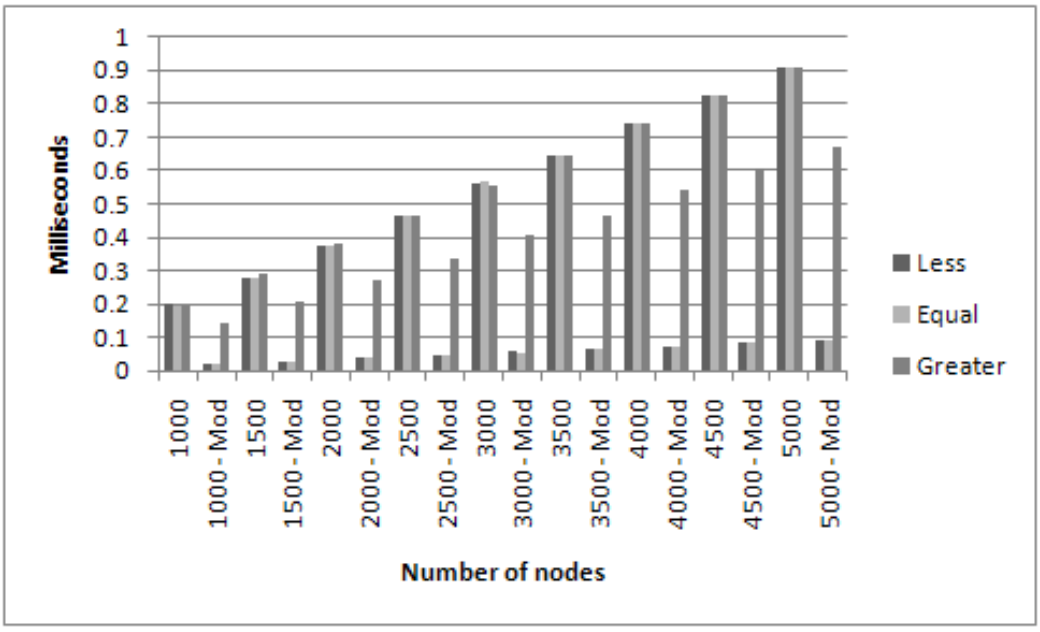


Figure 3.56: Decrease-key time complexity for standard and modified 3-4 heap

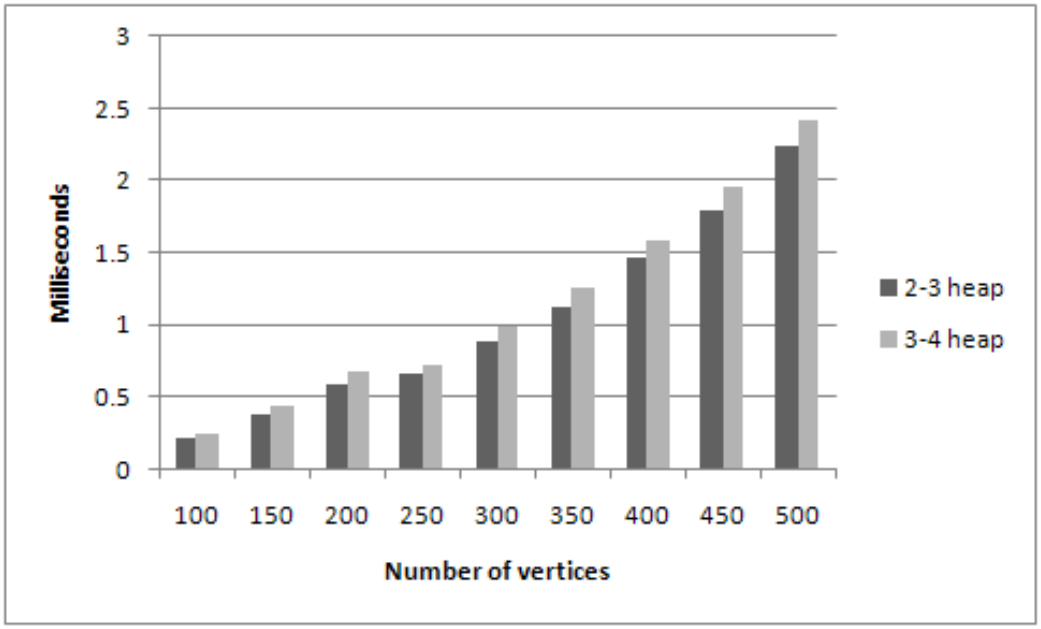


Figure 3.57: Dijkstra time complexity using standard 2-3 heap and 3-4 heap

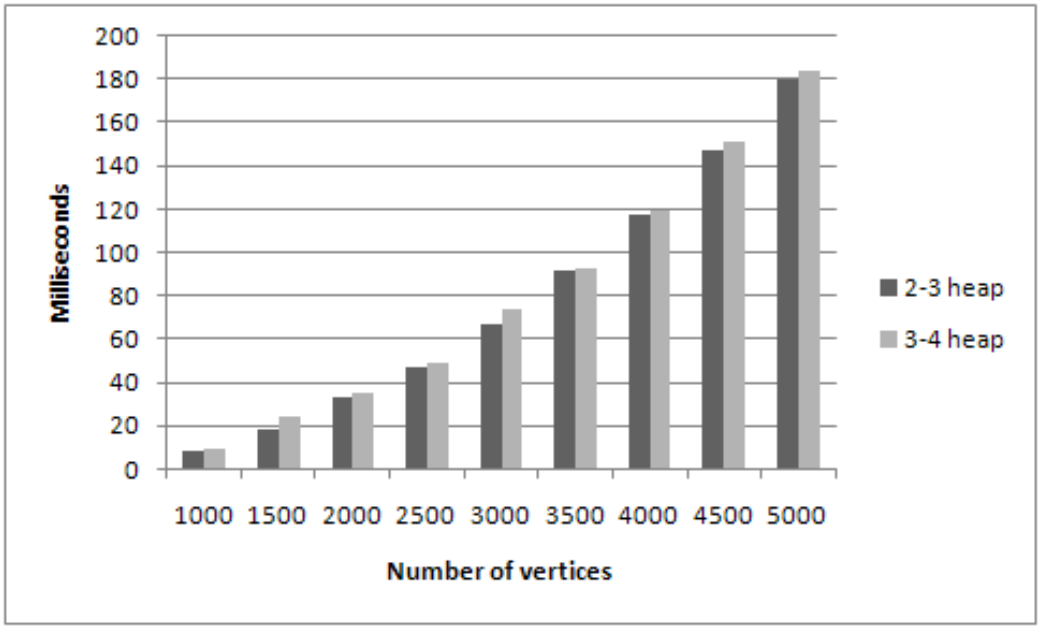


Figure 3.58: Dijkstra time complexity using standard 2-3 heap and 3-4 heap

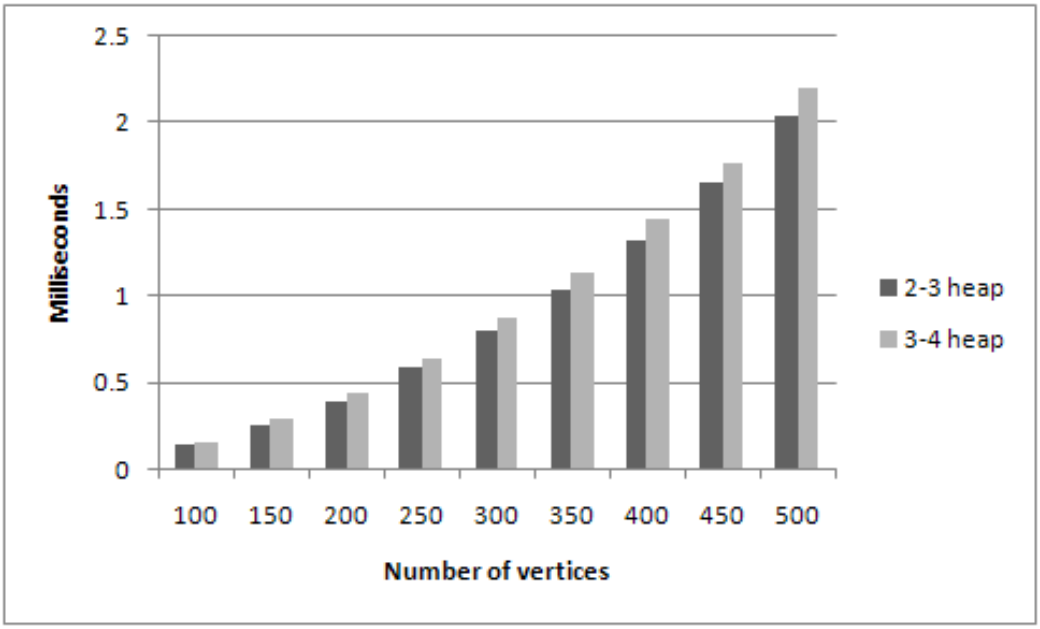


Figure 3.59: Dijkstra time complexity using modified 2-3 heap and 3-4 heap

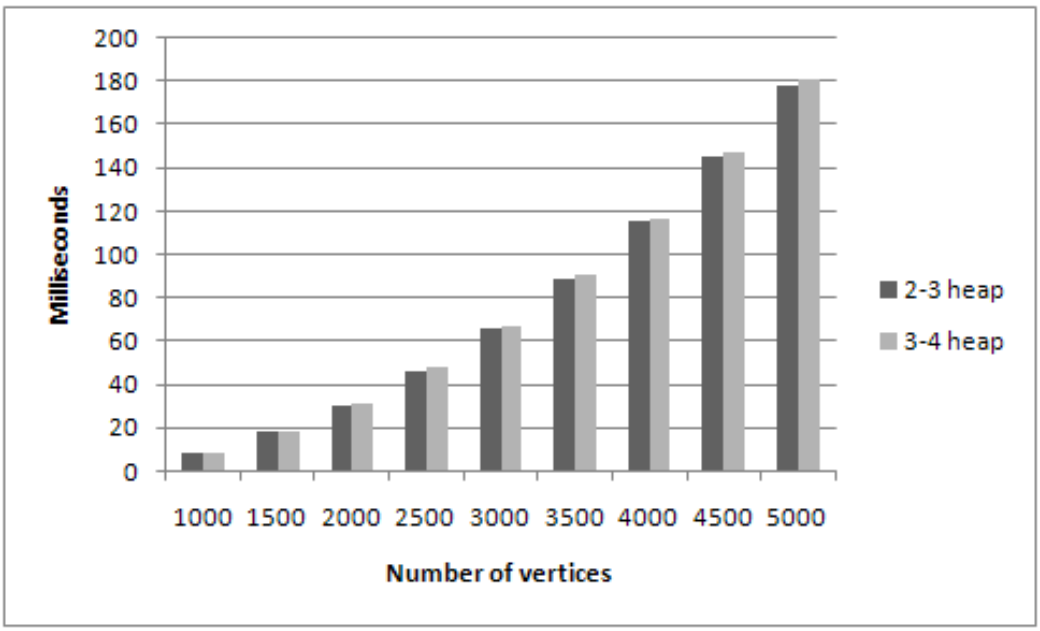


Figure 3.60: Dijkstra time complexity using modified 2-3 heap and 3-4 heap

| Number of Nodes | Insert | | | | Delete-min | |
|---|---|---|---|---|---|---|
| | Standard | | Modified | | | |
| | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap |
| 100 | 0.018 | 0.019 | 0.015 | 0.015 | 0.011 | 0.009 |
| 150 | 0.022 | 0.023 | 0.018 | 0.017 | 0.013 | 0.012 |
| 200 | 0.029 | 0.030 | 0.024 | 0.022 | 0.018 | 0.017 |
| 250 | 0.036 | 0.046 | 0.029 | 0.030 | 0.021 | 0.020 |
| 300 | 0.043 | 0.046 | 0.034 | 0.035 | 0.026 | 0.028 |
| 350 | 0.050 | 0.050 | 0.040 | 0.040 | 0.025 | 0.022 |
| 400 | 0.056 | 0.060 | 0.045 | 0.045 | 0.026 | 0.025 |
| 450 | 0.063 | 0.065 | 0.051 | 0.049 | 0.030 | 0.028 |
| 500 | 0.071 | 0.069 | 0.056 | 0.051 | 0.034 | 0.031 |
| 1000 | 0.163 | 0.145 | 0.139 | 0.111 | 0.095 | 0.082 |
| 1500 | 0.223 | 0.212 | 0.184 | 0.163 | 0.135 | 0.124 |
| 2000 | 0.288 | 0.290 | 0.239 | 0.218 | 0.137 | 0.125 |
| 2500 | 0.354 | 0.355 | 0.289 | 0.274 | 0.177 | 0.157 |
| 3000 | 0.430 | 0.422 | 0.345 | 0.325 | 0.219 | 0.189 |
| 3500 | 0.490 | 0.495 | 0.397 | 0.374 | 0.258 | 0.219 |
| 4000 | 0.561 | 0.571 | 0.450 | 0.468 | 0.298 | 0.252 |
| 4500 | 0.631 | 0.636 | 0.516 | 0.513 | 0.319 | 0.288 |
| 5000 | 0.743 | 0.700 | 0.570 | 0.540 | 0.375 | 0.322 |

Table 3.9: Insert and delete-min time complexity for Figures 3.47–3.50 and Figures 3.51–3.52

| Number | Standard Heap | | | Modified Heap | | |
|---|---|---|---|---|---|---|
| of Nodes | Less | Equal | Greater | Less | Equal | Greater |
| 100 | 0.012 | 0.011 | 0.011 | 0.003 | 0.003 | 0.010 |
| 150 | 0.027 | 0.022 | 0.022 | 0.005 | 0.005 | 0.019 |
| 200 | 0.022 | 0.033 | 0.030 | 0.006 | 0.005 | 0.023 |
| 250 | 0.034 | 0.037 | 0.034 | 0.007 | 0.007 | 0.027 |
| 300 | 0.042 | 0.041 | 0.041 | 0.007 | 0.008 | 0.032 |
| 350 | 0.049 | 0.036 | 0.048 | 0.009 | 0.009 | 0.091 |
| 400 | 0.059 | 0.058 | 0.055 | 0.009 | 0.014 | 0.043 |
| 450 | 0.062 | 0.063 | 0.062 | 0.011 | 0.011 | 0.048 |
| 500 | 0.065 | 0.065 | 0.065 | 0.012 | 0.012 | 0.049 |
| 1000 | 0.100 | 0.099 | 0.096 | 0.018 | 0.018 | 0.078 |
| 1500 | 0.141 | 0.140 | 0.142 | 0.027 | 0.026 | 0.107 |
| 2000 | 0.188 | 0.192 | 0.188 | 0.034 | 0.035 | 0.147 |
| 2500 | 0.240 | 0.236 | 0.242 | 0.043 | 0.043 | 0.183 |
| 3000 | 0.284 | 0.283 | 0.289 | 0.052 | 0.052 | 0.215 |
| 3500 | 0.332 | 0.335 | 0.356 | 0.088 | 0.060 | 0.249 |
| 4000 | 0.376 | 0.381 | 0.371 | 0.070 | 0.069 | 0.283 |
| 4500 | 0.414 | 0.413 | 0.512 | 0.078 | 0.077 | 0.317 |
| 5000 | 0.469 | 0.473 | 0.476 | 0.087 | 0.086 | 0.354 |

Table 3.10: 2-3 heap decrease-key time complexity for Figures 3.53–3.54

| Number | Standard Heap | | | Modified Heap | | |
|---|---|---|---|---|---|---|
| of Nodes | Less | Equal | Greater | Less | Equal | Greater |
| 100 | 0.026 | 0.025 | 0.021 | 0.002 | 0.002 | 0.022 |
| 150 | 0.042 | 0.038 | 0.041 | 0.005 | 0.005 | 0.032 |
| 200 | 0.041 | 0.054 | 0.053 | 0.006 | 0.005 | 0.040 |
| 250 | 0.068 | 0.049 | 0.061 | 0.007 | 0.007 | 0.046 |
| 300 | 0.079 | 0.074 | 0.076 | 0.007 | 0.008 | 0.060 |
| 350 | 0.079 | 0.068 | 0.075 | 0.009 | 0.009 | 0.070 |
| 400 | 0.082 | 0.078 | 0.102 | 0.011 | 0.010 | 0.079 |
| 450 | 0.118 | 0.119 | 0.119 | 0.011 | 0.012 | 0.087 |
| 500 | 0.131 | 0.131 | 0.130 | 0.015 | 0.012 | 0.096 |
| 1000 | 0.198 | 0.194 | 0.191 | 0.019 | 0.019 | 0.145 |
| 1500 | 0.279 | 0.280 | 0.291 | 0.029 | 0.028 | 0.204 |
| 2000 | 0.375 | 0.376 | 0.378 | 0.039 | 0.039 | 0.271 |
| 2500 | 0.465 | 0.465 | 0.464 | 0.047 | 0.046 | 0.336 |
| 3000 | 0.559 | 0.567 | 0.557 | 0.056 | 0.054 | 0.409 |
| 3500 | 0.643 | 0.642 | 0.643 | 0.063 | 0.065 | 0.467 |
| 4000 | 0.739 | 0.741 | 0.740 | 0.072 | 0.072 | 0.541 |
| 4500 | 0.829 | 0.827 | 0.828 | 0.081 | 0.081 | 0.605 |
| 5000 | 0.911 | 0.909 | 0.908 | 0.091 | 0.089 | 0.673 |

Table 3.11: 3-4 heap decrease-key time complexity for Figures 3.55–3.56

| Number | Standard Heap | | Modified Heap | |
|---|---|---|---|---|
| of Vertices | 2-3 heap | 3-4 heap | 2-3 heap | 3-4 heap |
| 100 | 0.209 | 0.234 | 0.141 | 0.150 |
| 150 | 0.379 | 0.438 | 0.249 | 0.281 |
| 200 | 0.578 | 0.670 | 0.386 | 0.435 |
| 250 | 0.651 | 0.723 | 0.585 | 0.631 |
| 300 | 0.883 | 0.983 | 0.799 | 0.868 |
| 350 | 1.125 | 1.258 | 1.032 | 1.127 |
| 400 | 1.461 | 1.586 | 1.316 | 1.439 |
| 450 | 1.786 | 1.961 | 1.650 | 1.763 |
| 500 | 2.235 | 2.414 | 2.046 | 2.198 |
| 1000 | 8.207 | 8.702 | 7.858 | 8.139 |
| 1500 | 17.796 | 23.758 | 17.779 | 17.983 |
| 2000 | 32.401 | 34.496 | 30.236 | 30.695 |
| 2500 | 46.960 | 48.604 | 45.930 | 47.292 |
| 3000 | 66.740 | 73.176 | 65.794 | 66.926 |
| 3500 | 91.567 | 92.354 | 88.971 | 90.370 |
| 4000 | 116.992 | 119.605 | 115.701 | 116.911 |
| 4500 | 147.099 | 151.428 | 145.472 | 147.133 |
| 5000 | 180.495 | 184.526 | 178.525 | 181.030 |

Table 3.12: Dijkstra time complexity for standard and modified 2-3 heap and 3-4 heap for Figures 3.57–3.60

# Chapter IV

# About 3-4 Heap Implementation

This section details the implementation requirements, considerations and highlights some elegant solutions to tricky scenarios. A number of programming languages were considered for this research like Java [11], C++ [12] and C [13]. Since the implementation language must be a bare bones no frills language with a minimal to none overhead, this made programming language C ideally suited for implementing data structures and algorithms.

## *4.1 Open Source Library*

Since the 2-3 heap and 3-4 heap share a lot of similarities between themselves, permission was granted by Tadao Takaoka to use the existing 2-3 heap implementation written by Shane Saunders [6]. This source code is stored in the algorithm repository provided by The University of Canterbury [14]. There were two conditions governing the usage of this code. The first was to allow the 3-4 heap implementation written during the course of this research to be added into the The University of Canterbury algorithm repository and secondly there was sufficient degree of separation in implementation requirements between both heaps.

Preliminary investigations revealed that whilst some basic operations like heap initialisation and tear down required minimal to no change, the core functional areas of insert and decrease-key could not be retained whilst delete-min required only minor modifications to handle the 3-4 heaps fourth node. The print to console feature required minimal adaptation whilst its internal data integrity checker had plenty of scope to undergo enhancement in order to reach one hundred percent coverage. The class structure used to internally represent a node required some minor adaptation so it could support the additional trunk node used by the 3-4 heap.

The 2-3 heap source code was left unchanged with the exception of under-

going the equivalent modifications to its insert and decrease-key processes. Since the 3-4 heap is derived from the 2-3 heap, being able to use this previously created 2-3 heap implementation would allow experiment results to be comparable and highlight similarities and changes in characteristics.

### *4.2 3-4 Heap API Data Structure Signature*

The Application Programming Interface (API) exposed by the 3-4 heap shares an identical signature with the 2-3 heap, which allows both heaps to be used by a consuming application without the need for source code modifications. Another major advantage of having an identical API, both heaps can be dynamically interchanged at run time without the consuming application requiring recompilation. Since the need of recompilation isn't required, both heaps can be used in series with exactly the same data set and therefore experimental results generated will be comparable.

```
typedef struct heap_info {
    int (*delete_min)(void *heap);
    void (*insert)(void *heap, int node, long key);
    void (*decrease_key)(void *heap, int node, long newkey);
    int (*n)(void *heap);
    long (*key_comps)(void *heap);
    void *(*alloc)(int n_items);
    void (*free)(void *heap);
    void (*dump)(void *heap);
} heap_info_t;
```

Figure 4.1: API signature shared by 2-3 heap and 3-4 heap

Figure 4.1 lists the shared API signature used by both the 2-3 heap and 3-4 heap. This is the API which the consuming application is programmed against. Through this API, the consuming application is able to get the 3-4 heap to either perform an action upon a node or return some counters. This API supports:

1. Delete minimum key valued node

| Legend | Description |
|---|---|
| heap | Instance of 3-4 heap passed as API function parameter |
| node | Vertex (array index) of node to be positioned inside heap (consuming application responsible for ensuring this is correct) |
| key | The node's original key value |
| newkey | The node's new key value |
| n_items | Maximum number of nodes the 3-4 heap will hold |

Table 4.1: Descriptions of API parameter names used in Figure 4.1

2. Insert node into tree $T(0)$

3. Decrease the key value of a node

4. Return integer value representing number of nodes contained

5. Return integer value representing number of key comparisons incurred

6. Initialise for storage for '$n$' nodes

7. Tear down heap and release all allocated memory

8. Print to console all nodes and perform integrity check of internal data structure

Options 1–3 represent the three core functional areas of insert, delete-min and decrease-key. Option 4 returns an integer value representing the current number of nodes contained within the heap. Option 5 returns an integer value representing the number of key comparisons performed since the heap was initialised. Options 6–7 are to setup and tear down the heap. Option 8 will print to console all the nodes contained within the heap at each top level position tree. The output is whitespace indented so each sub-tree $\mathbf{b_j}$ $(j = 0, \ldots, i-1)$ of tree $\mathbf{a_i}T(i)$ can be readily identified. Behind the scenes of the print to console functionality is a very critical feature and that is the built-in integrity checker. No expense was spared on the integrity checker

to reach one hundred percent coverage in detecting inconsistent data. Should an inconsistency be detected, then application execution halted immediately with an error message. The built-in integrity checker was switched off during time complexity measurements by wrapping the source code in a compiler inclusion directive and ensuring this was set to exclude.

### *4.3 Structure of the Node*

The class structure used internally to represent each node not only must store its own vertex number and key value, but also a variety of information to support the 3-4 heap implementation. Figure 4.2 lists the structure used to represent a node and Figure 4.4 is a diagram showing how the different properties on this structure work together to create the 3-4 heap tree. But first let's explain what each structure property listed in Figure 4.2 is and its role within Figure 4.4:

```
typedef struct tfheap_node {
    struct tfheap_node *parent;
    struct tfheap_node *child;
    struct tfheap_node *left, *right;
    struct tfheap_node *partner;
    int partnerPosition;
    int partnerCount;
    int dim;
    long key;
    int vertex_no;
} tfheap_node_t;
```

Figure 4.2: Structure representing the 3-4 heap node

Property '*parent*' provides a link to the parent node of this node. The dimension value of the parent node will always be at least one (1) higher than this node's dimension. If the node has got no parent node, then this property is unused. Inspecting Figure 4.3, nodes which are linked to through the 'parent' property are highlighted with a double circle.

Property '*child*' provides a link to the child node of this node. The dimension value of the child node will always be one (1) less than this node's

dimension. If the node has got no child node, then this property is unused. The 'child' and 'parent' properties provide a circular linkage. The higher the dimension of this node, the more nodes located in lower dimensions which have their respective 'parent' property linking to this node. Since the 'child' and 'parent' properties provide a circular linkage but can only link two adjacent nodes, an alternative linkage path needs to be created. The alternative linkage path is created using the 'left' and 'right' properties and is discussed in the next paragraph. Inspecting Figure 4.3, nodes which are linked to through the 'child' property are highlighted by circles which are solid filled and nodes which have their 'child' and 'parent' properties linking to each other are highlighted with a thicker line. A node in one dimension can be both a child and a parent in relation to other dimensions.

Property '*left*' provides a link to a child located one (1) dimension lower than this node and the parent node of that node is the same as this node's parent. If this node has got no parent node, then this property is linked to this node. If this node is located in dimension zero (0), then the 'left' property will link to the node located one (1) dimension below this node's parent, the node linked through the 'child' property of this node's parent, which could even be itself. Traversing the 'left' property will result in a circular one way path across all nodes which have the same node as their parent node. The number of traversals steps required to make a full circuit is defined as this node's parent dimension value minus one.

Property '*right*' is identical to property 'left' property described above, except its linkages traverse in the opposite direction. These two properties create a two way circular linked list.

Property '*partner*' provides a one way circular linked list between two or three nodes located on the same trunk in the same dimension. This property does not link to the parent node of the trunk. This property links to the next positioned node on the trunk unless this node is at the end of the trunk in which case it will link to the trunks first partner node.

Property '*partnerPosition*' provides a quick lookup reference which indicates which partner position this node is located at. Valid values are one (1), two (2) and three (3). Inspecting this properties value provides a quick mechanism to determine how many traversals are required to reach the first

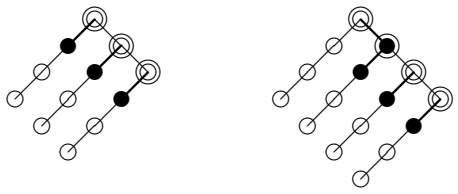

Figure 4.3: Two 3-4 heaps with their parent/child relationships highlighted

partner node.

Property '*partnerCount*' provides a quick lookup reference which indicates how many partners a node has. The value is only non-zero on the first partner node on a trunk and zero for all other nodes.

Property '*dim*' records the dimension of the node. The value is a non-negative integer.

Property '*key*' is the key value of the node. This value is controlled by the consuming application and is used by the 3-4 heap to determine a node's insertion position in ascending order of key value on a trunk.

Property '*vertex_no*' is the vertex number or index of the node. This value is controlled by the consuming application and must be unique among all nodes. The value is also used internally by the 3-4 heap as the array index where it maintains linkages to all nodes contained within the heap.

Of the afore mentioned properties, property '*partnerPosition*' and '*partnerCount*' are new to the 3-4 heap and replace '*extra*' as used by the 2-3 heap. In the 2-3 heap, property '*extra*' was used to hold a Boolean value indicating if the node was the third node on the trunk or not. For the 3-4 heap, additional granularity was required to identify not only the third node on the trunk, but the fourth node too. To achieve this, property '*partnerPosition*' was created to provide this additional granularity. Property '*partnerCount*' was created for performance improvement purposes because it is used as a quick reference lookup for determining how many nodes there are on the trunk. At the cost of increasing the memory footprint of each node, using this approach is quicker than walking the pointers and counting each node iterated over every single time the algorithm needs to know how many nodes are in the trunk.

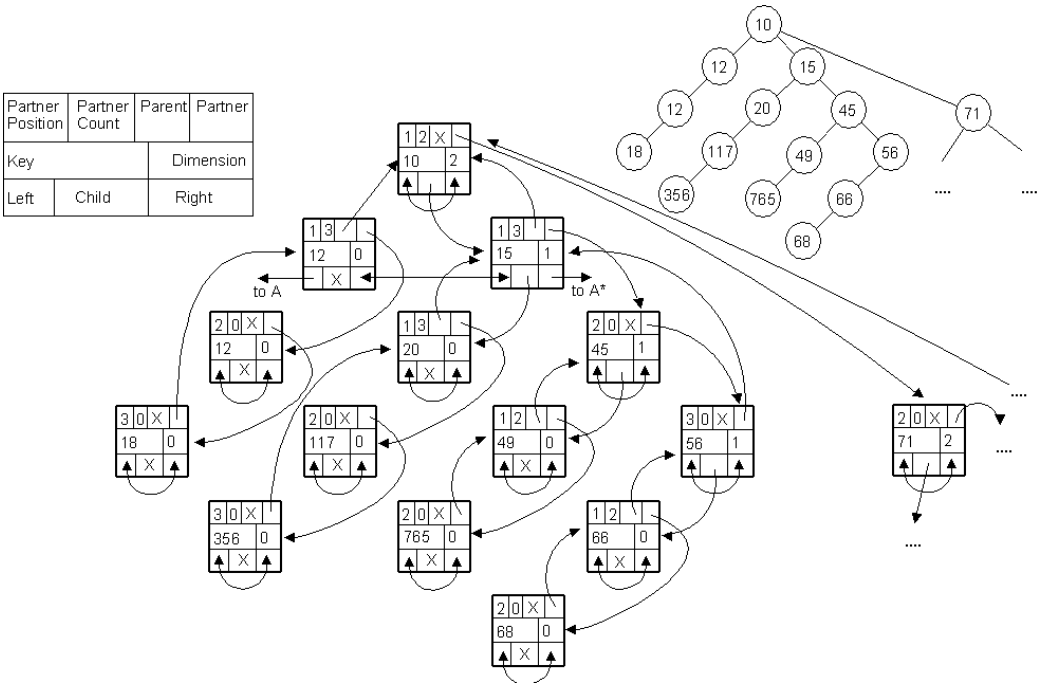


Figure 4.4: 3-4 heap internal representation of node connectivity

### *4.4 Insertion of a Node*

To ensure the decrease-key make-up process of case 14 in Section 2.8.1 was never greater than zero, we defined a four node trunk with a potential of six (6). The insertion process used to insert a single node into a three node trunk required at worst case three linear comparisons. To ensure an efficient implementation, the actual insertion process used to insert a single node into a three node trunk was algorithm $B$ in Section 2.7 because it consistently used two comparisons.

### *4.5 Coding Highlights*

When a trunk parent node is being relocated down one (1) dimension it will become a child of a neighbouring trunks parent node and its child node will become its partner, this changes many relationship linkages. This is a decrease-key scenario and is covered by cases three (3) and eleven (11) in Section 2.8.1. In Figure 4.5 left workspace, trunk $c$ is being relocated beneath

the parent of trunk $b$ and the other nodes of trunk $b$ will create the new trunk $c$. The right workspace presents the same workspace after the parent node has been relocated one dimension down.

The simplest linkage relationship to change is that of parent and child into partners but the more difficult linkage relationship to update is the reconnection of its child which is located two (2) dimensions lower as its new child because the 'left' and 'right' properties are used to create a circular linked list with other nodes that share the same parent node. This 'left' and 'right' circular linked list is identified in Figure 4.4 as '$toA$' and '$toA$*' on the two nodes located immediately beneath the root node.

Throughout the detailing of the 3-4 heap theory, we have stressed that we only take into consideration nodes within our current workspace for dimensions $i$-th and $(i+1)$-th, however, for implementation purposes there is a requirement to visit nodes in the $(i-1)$-th dimension during the course of an operation. Fortunately there is no requirement to go lower than the $(i-1)$-th dimension and it is only the scenarios which handle the creation and destruction of this parent/child relationship which do visit the $(i-1)$-th dimension.

This parent/child relationship is highlighted in Figure 4.5 in the left workspace where the parent node is identified by having a double circle and its child located two (2) dimensions lower is solid filled, it is the solid filled node which is located in the $(i-1)$-th dimension. In order to change this relationship into that of a parent and child, the following steps are required:

1. Remove current child node labelled '2' from the 'left' and 'right' circular linkage

2. Set parent node labelled '1' 'child' property to the 'left' property of its just removed child node labelled '2'. This will make the child node labelled '3' which is located two (2) dimensions lower, the new child node

3. Set the 'left' and 'right' properties of the former child node labelled '2' to link onto itself

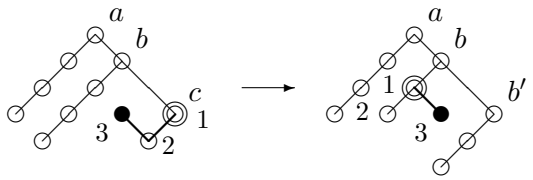


Figure 4.5: Left workspace represents a parent node and a child located two dimensions lower. Right workspace represents the same relationship after parent has been relocated one dimension lower

An exception to the above rules is when the child node is located in dimension zero (0) and the parent is in dimension (1). In this situation, the childs 'left' and 'right' properties point to itself and therefore there is no other node to reconnect as the child beneath the parent node.

The final workspace layout after completing these steps is highlighted in Figure 4.5 in the right workspace, where the parent node is highlighted by having a double circle and its child which used to be located two (2) dimensions lower is solid filled.

The following segment of source code achieves the reconnection of a trunk parent node with its child, if exists, located two (2) dimensions beneath itself:

```
if (childNode->left == childNode) {
   // Parent relocates to dimension zero
   parentNode->child = NULL;
}
else {
   // Re-link child located two dimensions beneath
   toReconnect = parentNode->child = childNode->left;
   toReconnect->right = childNode->right;
   toReconnect->right->left = toReconnect;
   childNode->left = childNode->right = childNode;
}
```

# Chapter V

# Future Research

During the course of this research, some areas were identified where the 3-4 heap could undergo further expansion. This section will bring these ideas into the light and are offered as areas where other researchers can actively pursue.

## *5.1 Delete a Node*

There is no dedicated 'delete' command on the API, so the only means in which to accomplish deletion is to use a combination of decrease-key and delete-min operations. This can be achieved by performing the following actions: first step is to reduce the node's key value to the smallest value it can represent. The second step is to perform a delete-min operation which will remove the node with the smallest key value. There is only one risk with this approach and that is the slightest possibility that another node has the same key value, in which case, the node located in the lowest tree $T(i)$ position will be removed and this may not necessarily be the desired node.

The core features required for deleting a node outright is for it to be removed from its current tree, that the tree remains within standard arrangement post this operation, and all sub-trees of this removed node are inserted into the appropriate $T(0),\ \ldots,\ T(i-1)$ tree. Before reinventing these processes, is there any existing functionality which can be leveraged off? In fact there is. The main features of decrease-key are as follows: reduce a node's key value, remove node from its tree, ensure its tree remains within standard arrangement, and finally insert the removed node into the appropriate $T(0),\ \ldots,\ T(i-1)$ tree. Having identified this, the 'delete' API can utilise actions two and three of the decrease-key process.

The last remaining step to be performed is the removal of all sub-trees $T(0),\ \ldots,\ T(i-1)$ beneath this node and inserting these sub-trees into the

appropriate $T(0),\ \ldots,\ T(i-1)$ tree. Before reinventing this process, is there any existing functionality which can be leveraged off? In fact there is. The main features of delete-min are as follows: locate node with smallest key value, remove sub-trees $T(0),\ \ldots,\ T(i)$ and insert these sub-tress into the appropriate $T(0),\ \ldots,\ T(i)$ tree. Having identified this, the 'delete' API can utilise the last two actions of the delete-min process. It is noted that delete-min supports a wider range of sub-trees, up to $T(i)$, against the 'delete' API requirement of up to $T(i-1)$. This greater range of sub-tree support isn't of concern because when a delete-min operation occurs, there could be only one node in $T(i)$ which will result in sub-tree $T(i)$ being empty.

### *5.2 Extended Insertion Cache*

The insertion cache, as noted in Section 3.3, is non-adaptive and under ideal conditions will automatically flush into top level position $T(1)$. The extended insertion cache takes this one step further by being able to hold trees of larger sizes because it does not automatically flush into $T(1)$ when full but instead commences to build a tree of size $T(2), \ldots,\ T(k-1)$. The aim of the extended insertion cache is to reduce the number of key comparisons required along an existing main trunk.

Should a delete-min operation occur, a modification must be made to this process so that the trees located within the cache are also scanned. This will result in a higher key comparison cost.

Should a decrease-key operation occur, modifications will be required to handle the scenario where the node having decrease-key performed on it is located within the cache. The node can either be located in the same sized $T(i)$ cache position or in a sub-tree $T(0),\ \ldots,\ T(i-1)$. This will have no impact on the key comparison cost incurred per decrease-key operation, but it will increase the implementation complexity and therefore increase the time complexity.

### *5.3 Decrease-key Cache*

When a decrease-key event occurs, the standard action is to immediately perform the four main actions, reduce the node's key value, remove a node

from its tree $T(i)$, ensure its tree is still within standard arrangement, and insert the removed node into the appropriate $T(0),\ \ldots,\ T(i-1)$ tree.

The aim of the decrease-key cache is to hold off from performing the final step, insertion. The removed node instead will be kept in a cache mechanism and once there are four nodes in the same sized $T(i)$ cache position, these will be flushed back into the heap.

Considerations for delete-min and decrease-key operations are discussed in Section 5.2.

### *5.4 Decrease-key Swap a Node*

When a decrease-key event occurs, the standard action is to immediately remove the node from its tree $T(i)$. A previously detailed modification was such that this node would remain in position provided its key value was equal to or greater than its immediate neighbour above. This modification will take this modification one step further because its aim is to prolong the duration a node remains within its current trunk. To achieve this, nodes on the same trunk will swap positions with their immediate or $2^{nd}$ neighbour above, and will only be removed once their key value is smaller than the trunk parent in $T(i+1)$.

The question to answer for this modification: Is swapping nodes within the same trunk less expensive than the combination of ensuring the tree $T(i)$ remains within standard arrangement and node reinsertion into the appropriate $T(0),\ \ldots,\ T(i-1)$ tree?

### *5.5 The 2-4 Heap*

Based upon the 2-3 heap, the 2-4 heap will have its maximum trunk length increased by one (1) to four (4) and the trunk will be permitted to shrink by two (2) nodes, thus retaining the minimum trunk length of two (2). This heap will inherit the improved delete-min performance of the 3-4 heap by having less top level workspace trees to scan, but retain the 2-3 heap decrease-key performance by having a minimum trunk size of two and its associated lower amortised cost when restructuring trunks back into standard arrangement.

It'll also inherit the improved performance that the 3-4 heap benefited from through the decrease-key modification and can likewise benefit from the future research modifications identified earlier in this section. With a maximum number of nodes in a workspace of sixteen (16) and minimum of four (4), the workspace buffer of the 2-4 heap will be twelve (12) nodes. Compare this to seven (7) for 3-4 heap and five (5) for 2-3 heap, and it'll be very interesting to see what kind of impact this larger buffer size will have on performance.

If the 2-4 heap implementation were to be based upon the 2-3 heap implementation written by Shane Saunders [6] in The University of Canterbury [14] algorithm repository, the following modifications are required:

1. Insertion of a node to be based upon the 3-4 heap implementation

2. Delete-min based upon the 3-4 heap

3. Modify the 2-3 heap decrease-key to support one additional node, plus new implementation code for supporting any new cases as required

4. Change the data structure representing a node instance like the 3-4 heap has been so that the trunk position of a node can be readily identified

Initially this research project was going to be on the 2-4 heap, but the number of possible decrease-key scenarios saw it change into the 3-4 heap. In hindsight, having a large number of decrease-key scenarios would have had the largest impact on the actual write up because implementation can take advantage of symmetry and therefore enjoys a reduced number of decrease-key permutations to support.

# Chapter VI

# Conclusion

This research has shown that under certain circumstances, it is possible for the 3-4 heap to outperform the 2-3 heap, especially when both heaps have undergone identical modifications. Under these conditions, the 3-4 heap experienced a performance improvement but the 2-3 heap performance worsened.

Having a workspace buffer two (2) nodes larger than the 2-3 heap, this would reduce the frequency of the expensive make-up processes and give a performance advantage. With amortised cost, the 2-3 heap had a consistent amortised cost of two during each decrease-key operation while the 3-4 heap was almost consistently three with several fours. When both heaps had not undergone modifications, this higher amortised cost was measurable because of the higher measured results during the insert and decrease-key operations.

With top level operations, the 2-3 heap had a consistent amortised cost of zero, a free operation, while the 3-4 heap varied between minus one and zero. This would give a performance advantage to the 3-4 heap across all three core operations of insert, delete-min and decrease-key, but by having a larger workspace buffer, the frequency of this event occurring would have been slightly reduced.

Inspecting the experimental results in Section 3.1 on the three main functions of insert, delete-min and decrease-key, the key comparison costs incurred were always slightly higher than the 2-3 heap with the exception of delete-min where they were always lower, as noted in Section 3.4.4. When both heaps underwent the insert and decrease-key process modifications, the 3-4 heap enjoyed a decrease in key comparison cost and it was possible for it to equal and even achieve a lower key comparison cost than the 2-3 heap. For the delete-min process, the 3-4 heap always enjoyed a lower key comparison cost because it had less top level positions occupied, whilst for decrease-key it

had an increase in key comparison cost because of its higher amortised cost. For experiments performed as the data store for Dijkstra's 'Single Source Shortest Path', the 3-4 heap performance was generally slightly worse than 2-3 heap except where the experiment used a densely connected graph with osculating distances, and then it always had the lowest key comparison cost.

Inspecting the experimental results in Section 3.7 on the sample experiments chosen for CPU time complexity analysis, these results indicated that there was a direct correlation between key comparison cost and time complexity required. Thus for experiments where the 3-4 heap achieved a lower key comparison cost than the 2-3 heap, it also achieved a lower time complexity.

One of the key lessons learned with this research is that increasing the length of the trunk didn't pay off very well but at the same time there wasn't a substantial degradation either. Having a workspace buffer which could hold two (2) more nodes than the 2-3 heap didn't make a detectable difference on performance by the reduced frequency of the expensive make-up process. But by increasing the minimum and maximum trunk lengths by one (1) to three (3) and four (4) respectively, there was a noticeable improvement in the delete-min performance but insert and decrease-key performance became worse.

The degrading of the insert and decrease-key performance wasn't all bad news because the 3-4 heap responded best to modifications. Along this line of thought, it would be very interesting to read the published research results from other computer scientists who investigated into the 2-4 heap and its performance relationship with the 2-3 heap and 3-4 heap.

As computers get faster and have more resources, the speed and efficiency of data structures is still a critical research area because the amount of data being processed increases such that it cancels any gain in hardware performance and the inverse of this is also true, where the same amount of data is being processed but the hardware resources available is limited, as found in hand held devices. Thus the demand for faster algorithms and more efficient data structures is more increasingly in high demand in the modern information age.

# References


[1] Takaoka, T., Theory of 2-3 Heaps, Discrete Applied Mathematics, Volume 126, 115–128, 2003.

[2] Dijkstra, E.W., A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.

[3] Prim, R.C., Shortest connection networks and some generalizations, Bell Sys. Tech. Jour. 36 (1957) 1389–1401.

[4] Fredman, M.L. and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Jour. ACM 34 (1987) 596–615.

[5] Vuillemin, J., A data structure for manipulating priority queues, Comm. ACM 21 (1978) 309–314.

[6] Saunders, S., A Comparison of Data Structures for Dijkstra's Single Source Shortest Path Algorithm, Department of Computer Science, University of Canterbury, 1999.

[7] Bethlehem, T., 3-4 Heap Workspace Operations, 5th New Zealand Computer Science Research Student Conference, April 2007

[8] Paul E. Black, Dictionary of Algorithms and Data Structures `http://www.nist.gov/dads/` [Online; accessed 27 February 2008], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004.

[9] Wikipedia, Dijkstra's algorithm, `http://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=200224965` [Online; accessed 30-March-2008], 2008.

[10] Wikipedia, Fibonacci heap, http://en.wikipedia.org/w/index.php?title=Fibonacci_heap&oldid=252723923 [Online; accessed 04-December-2008], 2008.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass., 2000.

[12] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 1997.

[13] B. W. Kernighan and D. M. Ritchie. The C Programming Language. Prentice Hall, Inc., 1978.

[14] Open Source, Algorithm Repository Home Page, University of Canterbury, http://www.cosc.canterbury.ac.nz/tad.takaoka/alg/rep.html [Online; accessed 30 November 2006].

[15] Takaoka, T., Lecture Notes on Graph Algorithms, University of Canterbury, http://www.cosc.canterbury.ac.nz/tad.takaoka/cosc229/graph.pdf [Online; accessed 06 April 2008].